Proyecto Fin de Carrera de Ingeniería en Informática

Ataques de relay en NFC con dispositivos Android

José Vila Bausili

Director: Ricardo J. Rodríguez Fernández

Ponente: José Javier Merseguer Hernáiz

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Septiembre de 2014
Curso 2013/2014

Agradecimientos

A mi familia por aguantarme, mantenerme y quererme.
A Arantxa por su paciencia y ánimos, no pocas veces.
A Ricardo, el mejor director de proyecto que podía haber tenido.
Y a José Merseguer, por hacerle trabajar en agosto.

GRACIAS.

On the Internet,
nobody knows you're a dog.
Abraham Lincoln

Ataques de relay en NFC con dispositivos Android

RESUMEN

Las siglas NFC (*Near Field Communication*) nombran al conjunto de estándares diseñados para establecer una comunicación inalámbrica punto-a-punto entre dispositivos en proximidad, normalmente de unos pocos centímetros. Dichos estándares cubren distintos protocolos de comunicación e intercambio de datos y están basados en otros estándares de identificación por radio frecuencia (RFID) como ISO/IEC 14443 o FeliCa.

Cada vez encontramos más servicios que permiten el pago mediante tarjetas o dispositivos sin contacto mediante tecnología NFC; desde el transporte público hasta aparcamientos, cajeros rápidos en supermercados o máquinas de *vending*. ¿El principal motivo? La fuerte apuesta de los bancos por esta tecnología.

Existen numerosos tipos de tarjetas NFC, de mecanismos de seguridad y de ataques a éstas. Los ataques de *relay* son una técnica de *man-in-the-midle* en la que el atacante es capaz de retransmitir un mensaje desde un emisor a un receptor remoto en tiempo real, explotando el supuesto de que la comunicación con una tarjeta NFC implica proximidad física. Desafortunadamente, la gran mayoría de tarjetas no dispone de ninguna medida ante este vector de ataque ya que la necesidad de hardware especializado "hace"poco realista un ataque práctico. Sin embargo, con la irrupción de dispositivos móviles con chips NFC, este panorama ha cambiado radicalmente.

En este trabajo se pretende estudiar la arquitectura NFC en un entorno móvil y desarrollar una aplicación llamada NFC Leech que permita realizar un ataque de *relay* con dispositivos Android a tarjetas NFC (concretamente tarjetas de crédito). Se pretende, a su vez, documentar todos los aspectos técnicos de la implementación que hace Android de NFC a partir de las especificaciones, documentación y código fuente, con el fin de servir de referencia a cualquier trabajo futuro relacionado.

Índice

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

Las siglas NFC (*Near Field Communication*) nombran al conjunto de estándares diseñados para establecer una comunicación inalámbrica entre dos dispositivos en proximidad, normalmente de unos pocos centímetros. Dichos estándares cubren distintos protocolos de comunicación e intercambio de datos y están basados en otros estándares de identificación por radio frecuencia (RFID), como ISO/IEC 14443 o FeliCa.

La tecnología RFID utiliza radio frecuencia para la transmisión de datos y nació con el objetivo de identificar y rastrear etiquetas añadidas a objetos, al igual que los códigos de barras. Estas etiquetas, o *tags*, almacenan información electrónica que es leída (o escrita) por los lectores y presentan una serie de ventajas respecto a los viejos códigos de barras como la no necesidad de visión directa del lector, la posibilidad de re-escritura o la mayor capacidad de almacenamiento.

Sin embargo su uso ha ido más allá del seguimiento y la gestión de *stock* en el sector industrial, hoy en día se puede encontrar en sistemas de autenticación, pago electrónico o sanidad, y sus aplicaciones no dejan de crecer. Esta gran difusión ha hecho que los teléfonos inteligentes no tarden en añadir chips e implementar capacidades para comunicarse mediante NFC, lo que ha supuesto una completa revolución en las aplicaciones móviles y en las formas de interaccionar con éstas.

1.1. Objetivo

Los objetivos de este PFC son: a) realizar un estudio de los ataques de *relay* en NFC –una variante de *man-in-the-midle* en la que el atacante es capaz de retransmitir un mensaje desde un lector (o PCD) hasta una tarjeta (o PICC) remota en tiempo real–; b) repasar la arquitectura NFC en dispositivos móviles (concretamente en Android); y c) documentar las posibilidades y limitaciones que implementan los fabricantes con el fin de establecer el estado del arte en los ataques de *relay* con dispositivos móviles. Además, se va a diseñar e implementar una aplicación Android que lleve a cabo un ataque de *relay* NFC real con tarjetas de crédito, utilizando dispositivos Android sin modificar, es decir, con hardware y software por defecto.

El procedimiento para obtener estos resultados será:

- Estudiar protocolos y estándares NFC para comprender su funcionamiento y familiarizarse con la literatura.
- Estudiar los ataques de *relay* a NFC, analizando resultados y limitaciones de los trabajos previos.
- Estudiar la API que ofrece Android para trabajar con NFC.
- Adquirir conocimientos necesarios de la arquitectura y diseño de Android para poder leer su código fuente (Java, C/C++) y seguir la implementación de la API NFC.
- Aprender a programar en Android para desarrollar la aplicación.

Y por último, presentar la investigación y las conclusiones de la manera más clara y formal posible.

1.2. Motivación

El gran crecimiento de sistemas que utilizan NFC hace interesante cualquier investigación que pueda afectar la seguridad de éstos. Aunque se han realizado trabajos sobre ataques de *relay* en dispositivos móviles (se estudian en detalle en el Capítulo 5), existe un vacío en cuanto a literatura sobre Android y NFC; posiblemente debido a su novedad y a los constantes cambios que ha sufrido en poco tiempo.

Hasta ahora, cualquiera que quisiera introducirse en el tema y entender cómo implementa Android la tecnología NFC necesitaba invertir una suma considerable de tiempo y esfuerzo buceando entre especificaciones, código fuente y foros. Este proceso de inmersión es necesario para intentar desarrollar cualquier herramienta avanzada o, simplemente, para entender el porqué de lo que se puede y no se puede hacer.

Se pretende que esta aportación pueda servir de referencia a todos los que quieren investigar a fondo la tupla NFC-Android, allanando así el camino a futuras mejoras o ataques.

Por otro lado, con la demostración de que cualquier usuario armado simplemente con la aplicación aquí presentada podría realizar este ataque, se persigue enviar un mensaje de concienciación que ayude a detectar y prevenir estos nuevos ataques y fraudes.

1.3. Organización

El documento está dividido en 6 capítulos y pretende seguir un orden que permita una lectura continua.

El Capítulo 2 define algunos conceptos previos que servirán al lector para comprender el resto del documento. Incluye una pequeña introducción a la familia de protocolos NFC, funcionamiento y nomenclatura en los ataques de *relay* y una introducción al entorno Android.

A continuación, en el Capítulo 3 se explica el diseño y la implementación NFC en Android: los distintos fabricantes e implementaciones, funcionamiento de las APIs y arquitectura, con el fin de analizar las posibilidades y limitaciones que se ofrecen para llevar a cabo un ataque de *relay*.

En el Capítulo 4 se documenta el diseño e implementación de la aplicación NFC Leech, capaz de realizar un ataque de *relay* NFC en Android en un escenario real de pago con tarjetas de crédito.

Para finalizar, en el Capítulo 5 se lleva a cabo una recopilación cronológica de los trabajos relacionados en el que se comparan sus resultados y limitaciones con los obtenidos en este trabajo.

Por último el Capítulo 6 incluye las conclusiones del trabajo y posibles líneas futuras de investigación.

Además el proyecto incluye una serie de apéndices con contenido más concreto, procedimientos llevados a cabo durante el desarrollo del proyecto o tablas y registros con más detalles:

- El Apéndice A incluye el diagrama de Gantt con las horas dedicadas a las diferentes partes de este proyecto.
- El Apéndice B incluye algunas tablas extraídas de las especificaciones del NFC Forum con detalles sobre los distintos parámetros de configuración del chip.
- El Apéndice C describe el flujo de llamadas de una invocación al método `transceive` de la API de Android.
- El Apéndice D documenta el proceso seguido para conseguir habilitar los *flags* de depuración del servicio NFC del sistema.
- El Apéndice E es una introducción a la programación de módulos para el Kernel de Android.
- El Apéndice F describe la estructura de los mensajes APDU, junto con las órdenes más utilizados.
- Por último, el Apéndice G incluye un gráfico extraído de `www.open-nfc.org` con las distintas tarjetas y tecnologías NFC.

Capítulo 2

Conocimientos previos

En este capítulo se definen conceptos básicos para la compresión de la memoria del trabajo. Dado que se abarcarán varias temáticas distintas y son necesarias bastantes nociones, se pretende dar una introducción a las distintas por secciones: empezando con la familia de protocolos y estándares que utiliza NFC, la explicación de en qué consiste un ataque de *relay* y finalmente se presenta el entorno Android para que el lector se familiarice con la nomenclatura.

2.1. Tecnología NFC y familia ISO 14443

La tecnología NFC o *Near Field Communication*, engloba a la familia de estándares para establecer una comunicación inalámbrica entre dos dispositivos en proximidad. Dichos estándares cubren distintos protocolos de comunicación e intercambio de datos y están basados en otros estándares de identificación por radio frecuencia (RFID) como ISO/IEC 14443 o FeliCa. Puesto que FeliCa es utilizado principalmente en Japón, esta sección se va a centrar en la familia ISO 14443 que es el estándar en Europa y EEUU.

La característica más relevante de NFC es la capacidad de los elementos pasivos, una comunicación consta de dos componentes: el PCD (o Proximity Coupling Device), que tiene corriente (elemento activo) y actúa como lector/escritor, y el PICC (o Proximity Integrated Circuit Card), que es la tarjeta pasiva y cuyo chip se alimenta por inducción gracias al campo electromagnético que genera el lector (por lo que no necesita una fuente de alimentación interna).

Dentro de ISO 14443 se puede distinguir dos tipos de tecnologías: A y B, que a pesar de trabajar ambas en la frecuencia 13'56 MHz se diferencian en la modulación utilizada, la codificación y los protocolos de inicialización. El estándar consta de 4 partes:

1. **ISO/IEC 14443-1:** Describe las características físicas.
2. **ISO/IEC 14443-2:** Describe la potencia y señal de la radio frecuencia.
3. **ISO/IEC 14443-3:** Detalla los algoritmos de inicialización y anti-colisión.

En la Figura 2.1 se puede ver un diagrama del protocolo de selección de una tarjeta, donde en caso de activar varias tarjetas a la vez se realiza un algoritmo de anti-colisión. Las órdenes `REQA`, `ATQA`, etc., son simplemente un conjunto de bytes definidos en la especificación.

En resumen, el PCD (o lector) se encuentra haciendo *polling* enviando órdenes `REQA` al medio hasta que un PICC (o dispositivo pasivo) entra en el campo y es activado, momento en el que responderá con un `ATQA` y quedará a la espera de ser seleccionado.

El PCD ejecutará su algoritmo de anti-colisión en caso de que se detecten varias tarjetas cercanas en el mismo momento. El algoritmo consiste en ir enviando una máscara de bits sobre el identificador de la tarjeta, ya que todas las tarjetas tienen un ID único, de este modo los PICCs cercanos responderán sólo si la máscara coincide con su ID. En cada iteración se restringirá más la máscara hasta que sólo responda una tarjeta. Finalmente el PICC seleccionado responderá con un mensaje `SAK` determinando el tipo de tarjeta que es y los protocolos que soporta.

4. **ISO/IEC 14443-4:** Protocolo de transmisión.

   En esta parte se define el protocolo de transmisión, y como se aprecia en la Figura 2.2, se produce el envío de unas órdenes de configuración (`RATS` y `PPS`) que definen los parámetros entre ambas partes para establecer un canal lógico de comunicación *half-duplex*, es decir, donde sólo puede estar transmitiendo un par a la vez.

   Dependiendo del `SAK` recibido, el PCD solicitará mediante una orden `RATS` (*Request ATS*) los parámetros de la conexión del *tag*. La respuesta `ATS` (*Answer To Select*) contiene por ejemplo los tamaños de ventana, los *timeouts* máximos para los mensajes de configuración y el canal, bytes históricos...

   Opcionalmente el PCD podrá modificar algún parámetro si la tarjeta soporta `PPS`. Una vez finalizado queda definido el canal lógico sobre el que se enviarán mensajes APDU.

Las tarjetas que cumplen las cuatro partes del estándar se denominan IsoDep, aunque también hay algunas que implementan protocolos propietarios y siguen algunas o ninguna parte. Un ejemplo son las tarjetas Mifare Classic, que implementan las dos primeras acorde al estándar, pero después establecen su propio protocolo de transmisión.

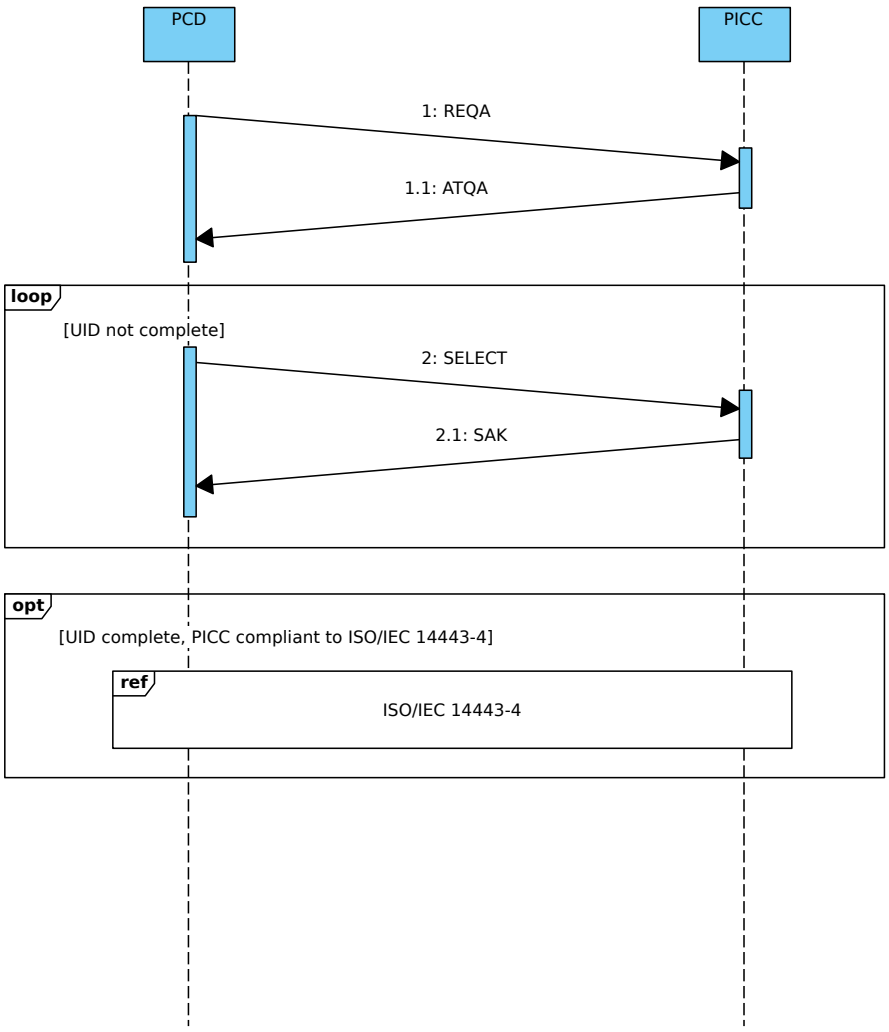


Figura 2.1: Protocolo de selección y anti-colisión ISO 14443-3A.

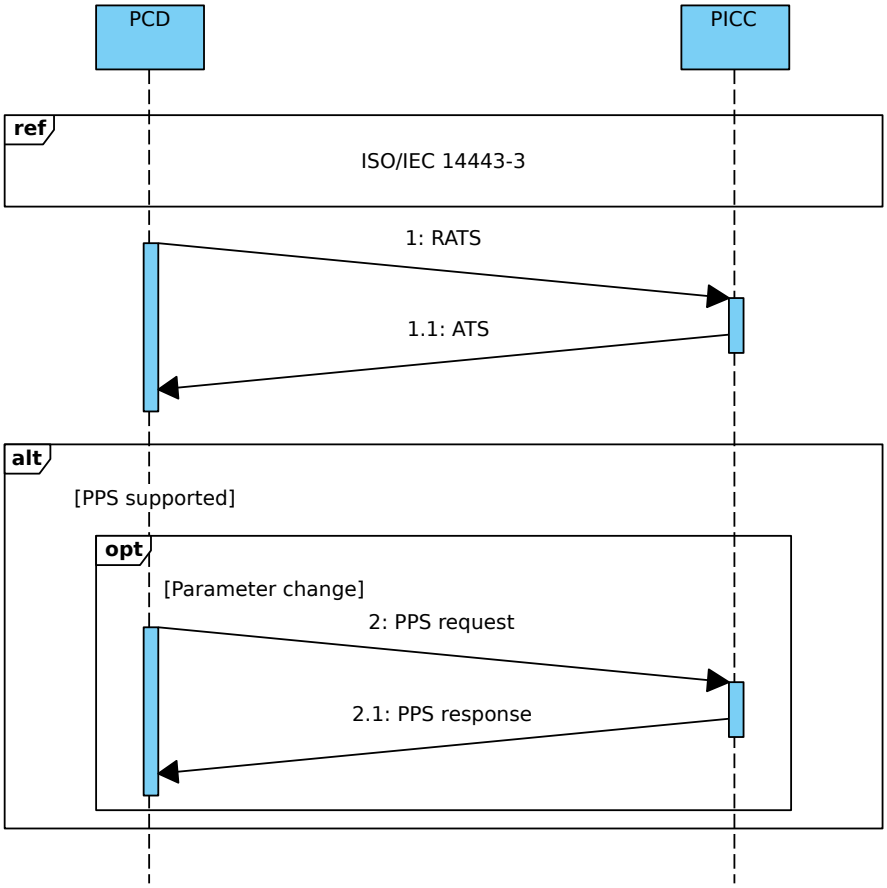


Figura 2.2: Preámbulo del protocolo de transmisión ISO 14443-4.

2.2. APDUs (ISO 7816-4) y EMV

El estándar ISO 7816 está relacionado con las tarjetas inteligentes o *smartcards* y consta de quince partes o libros. Para este trabajo será de interés la parte 4 donde se define la organización y seguridad de las órdenes para intercambiar información, y que puede ser usada independientemente del medio físico.

NFC utiliza los mensajes APDU o Application Protocol Data Unit (especificados en ISO 7816-4), que es la estructura de los mensajes transmitidos sobre el canal *half-duplex* establecido en base a ISO 14443-4. En las Figuras 2.3 y 2.4 se pueden ver los diagramas de envío de un mensaje APDU entre el PCD y el PICC.

Puesto que ISO 7816 define un protocolo orientado a tarjetas inteligentes existe una orden especialmente relevante llamada `SELECT`. Esta orden permite a un lector seleccionar una aplicación concreta de la tarjeta con la que comunicarse, ya que los chips pueden almacenar y ejecutar distintas aplicaciones. Este proceso es posible gracias a los AID (o Identificadores de Aplicación) que están formados por dos partes:

- **RID** (o *Registered Application Provider Identifier*): consta de 5 bytes y está registrado por una autoridad registradora.
- **PIX** (o *Propietary Identifier Extension*): permite al proveedor diferenciar entre sus propias aplicaciones dentro de una misma tarjeta.

Por otro lado, el estándar EMV (de Europay, MasterCard y Visa)[1] define la comunicación entre tarjetas inteligentes y terminales TPV (Terminal Punto de Venta, o POS en inglés) o cajeros automáticos (ATM) para la autenticación en transacciones de tarjetas de crédito y débito. Las órdenes EMV utilizan la estructura APDU, que se puede consultar para mayor detalle en el Apéndice F.

[1] http://www.emvco.com/specifications.aspx?id=21

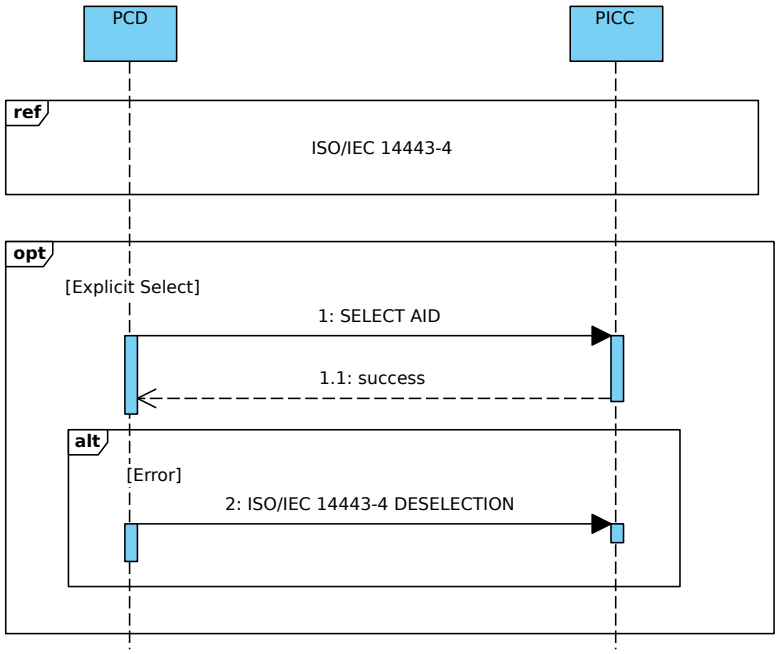


Figura 2.3: Selección opcional de AID en un canal ISO 14443-4.

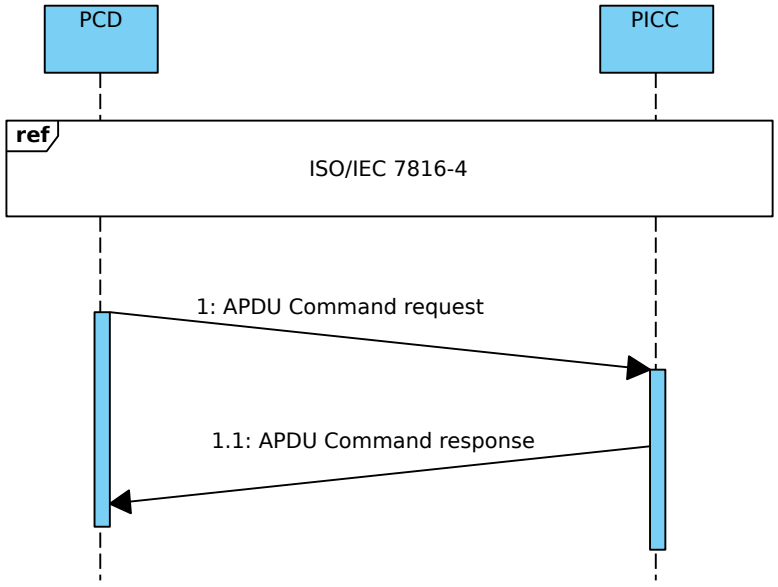


Figura 2.4: Envío de un APDU sobre ISO 14443-4.

2.3. Ataques de *relay*

Los ataques de *relay* son el tema central de este trabajo, de modo que se pretende introducir la idea subyacente y posteriormente presentar el caso concreto en NFC.

La primera vez que aparece este concepto fue en [CON76] en el año 1976, donde Conway explica cómo alguien que no conoce las reglas del ajedrez puede llegar a vencer a un Gran Maestro. El escenario consiste en retar a dos Gran Maestros a una partida postal, es decir por correo, y hacer el *relay* (retransmisión) de los movimientos entre ellos. La única restricción es que en una partida deberá jugar como Blancas y en la otra como Negras.

En la Figura 2.5 se muestra el escenario descrito por Conway, donde el caballo blanco representa el Gran Maestro que juega con blancas, y el negro a su rival. Aunque en realidad están jugando entre ellos, ambos creen que su rival legítimo es el señor del sombrero.

La aplicación de este ataque en protocolos de desafío-respuesta, comúnmente usados en transacciones y pagos, fue descrita por primera vez en [DGB87]. Una cronología posterior de los estudios relacionados con ataques de *relay* en NFC se puede consultar en el Capítulo 5.

El caso particular en NFC pretende explotar la confianza del PCD en que el *tag* legítimo se encuentra en su proximidad: el atacante se situará en medio de la comunicación suplantando al PICC de cara al PCD, y haciendo creer a la tarjeta que es el lector legítimo. La principal ventaja es que, al igual que el falso jugador de ajedrez, no es necesario conocer el protocolo de aplicación y cualquier medida de seguridad a ese nivel (como el cifrado) resultará inútil.

La principal limitación de estos ataques es el *delay* o retraso introducido durante el *relay*, ya que algunos protocolos poseen restricciones temporales bastante estrictas.

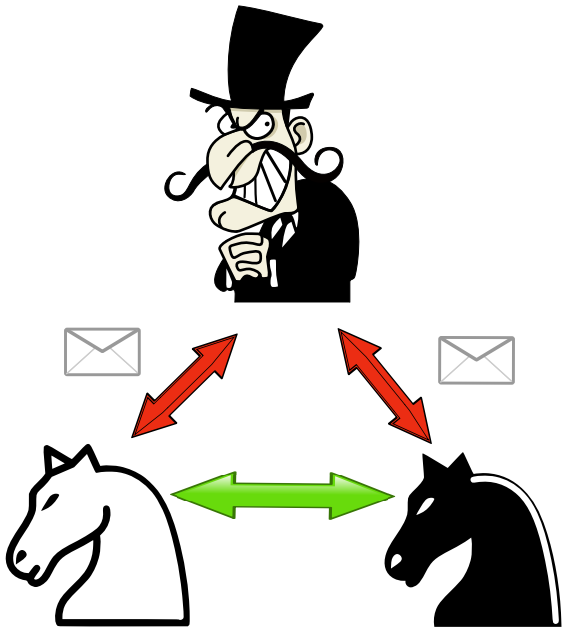


Figura 2.5: Escenario de *relay* en partida de ajedrez postal descrito por Conway.

2.4. Ecosistema Android

Android es un sistema operativo móvil basado en el Kernel de Linux, actualmente desarrollado por Google y que desde 2011 posee la mayor cuota de usuarios del mercado, siendo instalado en más dispositivos que Windows, iOS y Mac OS juntos. Como consecuencia en los últimos años se ha creado todo un ecosistema a su alrededor. En esta sección se van a describir algunos conceptos clave para dar a conocer este entorno al lector.

2.4.1. Dispositivos y software

La arquitectura hardware principal de Android es ARMv7 de 32 bits (aunque posteriormente se ha dado soporte a otras arquitecturas como x86 o MIPS), pero al ser un sistema utilizado por un conjunto enorme de dispositivos de diferentes fabricantes, con distintos componentes como acelerómetros, giroscopios, cámaras, pantallas táctiles, etc., han ido apareciendo diferentes ramas del sistema mantenidas ya sea por los propios fabricantes o por comunidades de desarrolladores.

El repositorio principal y la base del sistema operativo se encuentra en `https://source.android.com` y está mantenida por el AOSP (*Android Open Source Project*) que es liderado por Google. Android ha ido evolucionando con los años, desde el lanzamiento de una versión beta en 2007 hasta la última versión 4.4.4 lanzada el 19 de junio de 2014. Normalmente los dos primeros números indican la versión mayor y suelen tener un nombre; la 4.4, por ejemplo, se llama *Kitkat*.

A partir de estos lanzamientos los fabricantes suelen lanzar versiones propias o ROMs, que son las imágenes del sistema que se instalan en el dispositivo con soporte para componentes particulares y *drivers* propios. También existen modificaciones de Android no oficiales que dan la posibilidad al usuario de utilizar funcionalidades y características especiales, entre las más conocidas están: CyanogenMod, AOKP, OmniROM o Paranoid Android.

Otra parte fundamental del ecosistema Android es su *market*, también conocido como Play Store, donde los desarrolladores pueden subir sus aplicaciones y los usuarios instalarlas muy fácilmente. Hay que mencionar que también existen *markets* no oficiales, como AppBrain o GetJar.

2.4.2. Desarrollo en Android

El desarrollo en Android es uno de los puntos fuertes del éxito del sistema, ya que a pesar de soportar dispositivos tan diferentes ofrece un SDK (Software Development Kit) y un *framework* con APIs de desarrollo muy bien documentadas[2] e independientes de la plataforma destino. El lenguaje utilizado es Java aunque a bajo nivel no utiliza su máquina virtual (JVM), sino Dalvik, una máquina virtual especialmente diseñada para la arquitectura Android.

A la par que las versiones del sistema operativo, la SDK y la API han ido evolucionando y añadiendo nuevas funcionalidades. Para hacer referencia a la versión de la API se habla de niveles, y por ejemplo con Android 4.4 se soporta el *Nivel de la API* 19. El concepto de nivel permite que un dispositivo pueda utilizar los niveles de API inferiores a su versión (se mantiene la compatibilidad hacia atrás), pero no los superiores.

Otro elemento clave es el Manifest de una aplicación, un fichero XML donde el desarrollador debe definir los distintos elementos de ésta, los permisos que va a requerir y los eventos que quiere ser capaz de escuchar. De este modo, cuando una nueva aplicación es instalada el sistema puede leer este fichero y registrarla correctamente.

2.4.3. Arquitectura Android

Como se ha visto Android está basado en el Kernel de Linux, aunque actualmente poco tiene que ver el uno con el otro. Las principales diferencias son la implementación de IPC (*Inter Process Communication*) en Android y el énfasis puesto en mejorar el consumo para ahorrar batería (lo que es crítico en dispositivos móviles). Además Android ofrece un entorno aislado (o *sandbox*) durante la ejecución de aplicaciones para proteger los datos del usuario, junto con un gestor de permisos que permite a una aplicación utilizar los distintos componentes del sistema (ficheros, cámara, USB, Internet, etc.).

A continuación se describe brevemente cómo implementa Android los servicios del sistema y cómo permite a las aplicaciones de usuario comunicarse con éstos.

Para aclarar, los servicios del sistema (79 en la versión 4.4) implementan la mayoría de las funcionalidades del núcleo de Android, muchas de ellas escritas en Java aunque

[2] https://developer.android.com/preview/setup-sdk.html

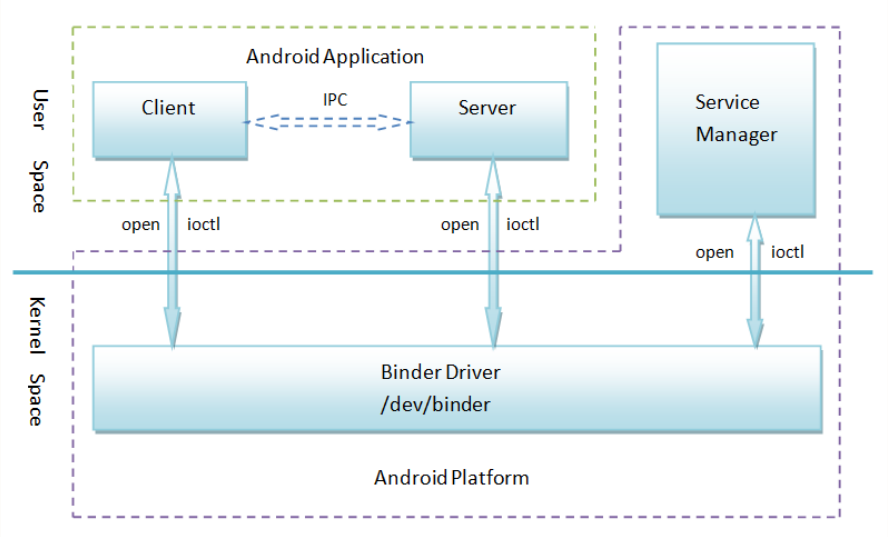


Figura 2.6: Arquitectura IPC basada en `Binder` de Android.

algunas partes pueden escribirse en código nativo gracias a JNI (Java Native Interface), un *framework* que permite a los programas escritos en Java interactuar con programas escritos en otros lenguajes como C, C++ o ensamblador.

Los servicios definen una interfaz remota que puede ser llamada desde otros servicios o aplicaciones, constituyendo así un sistema operativo orientado a objetos sobre Linux. La base de este mecanismo es el gestor de servicios, que será el encargado de gestionar el registro y descubrimiento de éstos, y de permitir la posterior comunicación mediante IPC.

Existen diversos mecanismos de comunicación entre procesos; incluyendo ficheros, señales, *sockets*, *pipes*, semáforos o memoria compartida, pero la solución de Android pretende ser más flexible y transparente. La implementación que hace Android está inspirada en OpenBinder, y se basa en la clase `Binder` para construir una arquitectura distribuida (véase Figura 2.6) de interfaces abstractas implementando un mecanismo similar a CORBA, aunque permitiendo tan sólo comunicaciones entre procesos y no a través de la red.

A nivel interno el *driver* se encarga de alojar los datos transmitidos en la memoria de los distintos procesos de forma segura.

Con todo esto se consiguen abstracciones de más alto nivel como los *Intents*, que son órdenes con datos asociados que utilizan los procesos y aplicaciones para comunicarse, y sobre los que funciona todo el sistema de eventos de las aplicaciones Android.

Capítulo 3

Análisis de NFC en Android

La API de Android (actualmente versión 19) ofrece a los desarrolladores un amplio conjunto de clases, servicios e interfaces para trabajar con NFC. Concretamente se puede distinguir entre tres tipos de comunicación: lectura/escritura de un *tag*, HCE (*Host card Emulation*) y P2P (también conocido como Android Beam).

En el primer caso, cuando el sistema operativo descubre una tarjeta cercana, ya sea porque el servicio NFC está haciendo *polling* o porque se detecta un campo próximo (*low power target detection*), se crea un objecto `Tag` que es pasado a la actividad suscrita mediante un `Intent` con el campo `EXTRA_TAG`. Este objeto es inmutable y representa el estado de la tarjeta NFC en el momento de su descubrimiento. A partir del `Tag` se forma la interfaz `TagTechnology` que describe el tipo de tarjeta y permite abstraer el protocolo y la implementación interna. Desde este momento se hará referencia exclusivamente al protocolo ISO 14443-3A, por ser el protocolo más extendido y sobre el que se implementa normalmente la tecnología IsoDep que intercambia mensajes APDU, como se vio en el Capítulo 2.

El método principal de comunicación que permite enviar una orden (conjunto de bytes) al *tag* remoto y sobre el que todas las aplicaciones deberán construir su propio protocolo es `transceive(byte[])`. Sin embargo, aunque a nivel interno la SDK especifica un *flag* para poder determinar si la comunicación debe transmitirse literalmente o no, los fabricantes lo ignoran y, como la documentación ya avisa, añaden un CRC automáticamente además de no permitir algunas órdenes propias de la fase de anticolisión/selección. Esta restricción hace imposible enviar órdenes *raw*, es decir, sin que sean modificadas, y por lo tanto es imposible controlar a nivel software el principio de una comunicación o trabajar con tecnologías propietarias que no sigan la especificación al completo, como Mifare Classic.

De modo que el primer paso es averiguar en qué punto (de las distintas capas de la Figura 3.1) se lleva a cabo este control y estudiar si es posible enviar órdenes *raw* o no. En la Tabla 3.1 se enumeran las distintas posibilidades de cada nivel junto con sus características.

Por otro lado, para utilizar HCE o el móvil como elemento pasivo, la aplicación debe registrar en su *Manifest* el AID que pretende emular. De modo que cuando un lector que

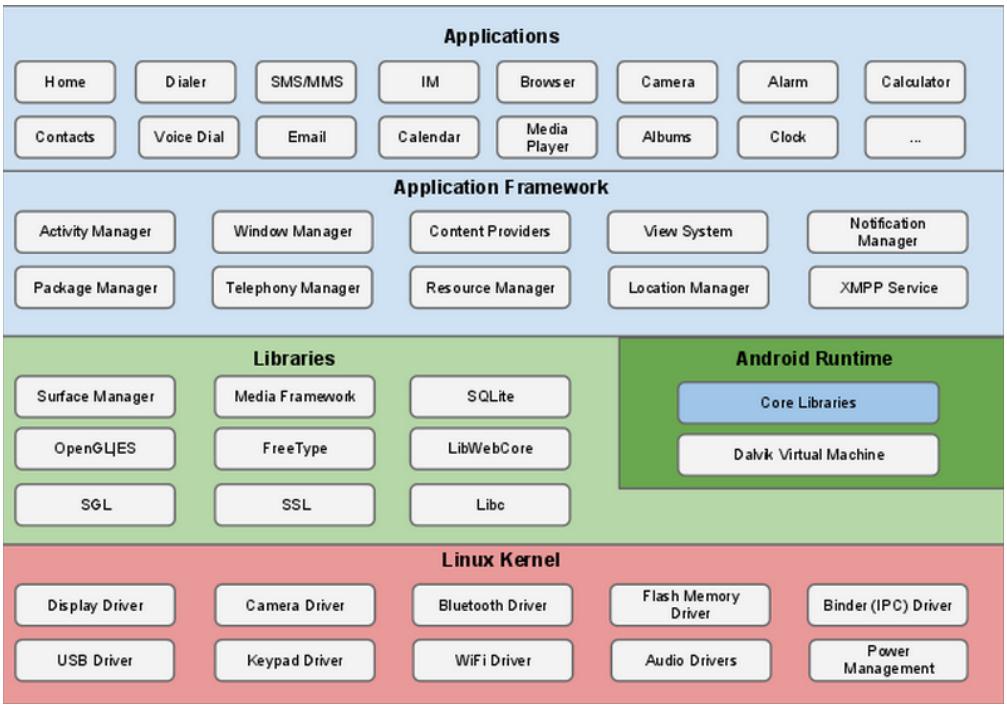


Figura 3.1: Capas de la arquitectura Android.

N.	Descripción	Lenguaje(s)	Dependencia	OpenSource
1	Paquete `com.android.nfc` `https://android.googlesource.com/platform/packages/apps/Nfc/`	Java, JNI y C++	Ninguna	Sí
2	Librería del sistema: `libnfc-nxp` o `libnc-nci` `https://android.googlesource.com/platform/external/libnfc-nci/`	C/C++	Fabricante	Sí
3	Kernel de Linux `https://android.googlesource.com/kernel/msm.git/+/master/drivers/nfc/`	C	Hardware y fabricante	Sí
4	Firmware Directorio `/system/vendor/firmware` del dispositivo.	Binario	Hardware y fabricante	No

Tabla 3.1: Tabla de niveles de la arquitectura NFC en Android.

trabaje en IsoDep envíe una orden o APDU `SELECT` especificando el AID de la aplicación (*smartcard*) con la que quiere comunicarse, el sistema creará una entrada en una tabla de enrutamiento[1] y los APDUs posteriores serán redirigidos a dicha aplicación (móvil); que deberá tener un servicio que extienda la clase `HostApduService` implementando el método `processCommandApdu`. A partir de ese momento, el sistema mantendrá el canal de comunicación hasta que llegue otra orden `SELECT` o la conexión termine.

Desafortunadamente, cómo se ha visto, no es posible *a priori* emular mediante software una tarjeta a nivel ISO 14443-3A, por lo que va a ser necesario estudiar si es posible evadir esta restricción en Android, y en caso de serlo, como llevarlo a cabo.

3.1. Alternativas y elección

El primer nivel de los que se muestran en la Tabla 3.1 tiene la ventaja de ser el mismo para cualquier dispositivo Android con soporte NFC, pero a partir de los siguientes dependerá completamente de la implementación.

En este trabajo se ha optado por realizar el análisis de la arquitectura NFC en un dispositivo LG Nexus 4 con chip Broadcom BCM20793 (el primero en usar un chip de esta familia), frente a los múltiples dispositivos con chip NXP PN544 de NXP Semiconductors como el Samsung Galaxy Nexus, que fue el primer móvil Android en incorporar soporte para la tecnología NFC.

La principal razón es que Broadcom apostó por crear su interfaz NCI (*NFC Controller Interface*) abierta, convirtiéndola en una nueva especificación del NFC Forum[2] para intentar estandarizar la abstracción entre el NFC Controller (software del chip o *firmware*) y el Device Host (sistema operativo del dispositivo móvil). Con esto se pretende dar más flexibilidad a los fabricantes de dispositivos para elegir un chip compatible con NCI, y más rapidez a los fabricantes de chips para introducirlos en el mercado (teniendo menos que implementar), por lo que se espera que ambas partes apuesten por NCI y su utilización crezca.

Otra motivo es que el PN544 no es completamente compatible con Android HCE (aunque el chip soporta el modo, la mayoría de ROMs no lo hacen y es necesario modificarla), lo que supone una restricción más a pesar de que actualmente se está trabajando para darle soporte por defecto. De hecho, como se ve en el Capítulo 5 CyanogenMod implementó la emulación software antes que el AOSP (Android Open Source Project), aunque exclusivamente para el chip de NXP.

El único inconveniente del chip Broadcom es la falta de soporte para algunas tarjetas de la familia Mifare [Sem11], que al usar un protocolo propietario sobre ISO 14443-3A y ser propiedad de NXP Semiconductors, tan sólo las soportan sus propios chips. Sin embargo, al estudiar en profundidad la implementación, se verá si es posible o no llegar a soportar estos protocolos.

[1] https://android.googlesource.com/platform/packages/apps/Nfc/+/android-4.4.1_r1/src/com/android/nfc/cardemulation/RegisteredAidCache.java

[2] http://nfc-forum.org/about-us/the-nfc-forum/

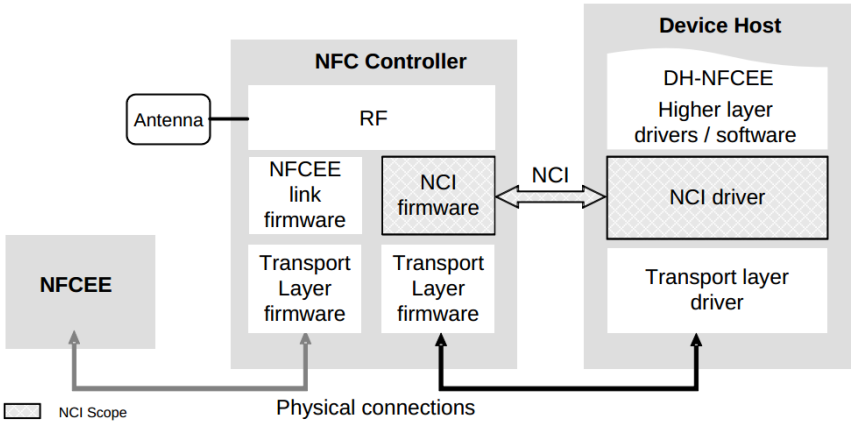


Figura 3.2: Arquitectura NCI: comunicación entre NFCC y DH.

3.2. NCI

El NFC Controller (NFCC) es el responsable de transmitir la información a través del canal de radio frecuencia, es parte del SoC o *system-on-chip*. Tiene una conexión al Device Host (DH), que en este caso es el sistema operativo del móvil, y puede habilitar otras conexiones adicionales a distintos entornos de ejecución (NFCEE), instalados por ejemplo en un SE (Secure Element). De este modo la NCI define la interfaz lógica entre el NFCC y el DH (puede verse en la Figura 3.2), y su principal objetivo es establecer un canal de comunicación fiable e independiente de la información transmitida, sobre cualquier canal físico (e.g., SPI, I2C, UART, USB, etc.). NCI define también la fragmentación de un mensaje en múltiples paquetes, dependiendo del MTU (o unidad máxima de transferencia) del medio.

La especificación define dos tipos de mensajes, de control y de datos, que serán empaquetados para transmitirse sobre un canal físico concreto:

- Los **mensajes de control** consisten en Comandos, Respuestas y Notificaciones, los Comandos sólo se envían desde el DH al NFCC, mientras que los otros dos pueden ir en ambas direcciones.
- Los **mensajes de datos** suelen llevar la información dirigida al NFC Endpoint (o *tag* remoto vía RF), y sólo se pueden enviar una vez se ha establecido una conexión lógica. Por defecto durante la inicialización se define el canal lógico a la conexión RF, pero pueden crearse canales adicionales hacia otros NFCEE (*NFC Execution Environment*). En cualquier caso, a partir de ahora y por simplificar, se ignorará la parte de conexión a otro entorno de ejecución.

Otro elemento clave son los módulos NCI, donde cada uno puede ser una Interfaz RF que define cómo el DH se va a comunicar a través de NCI con un NFC Endpoint. Cada Interfaz RF soporta un protocolo específico y define como el *payload*, o contenido, de un

mensaje de datos NCI encaja en el *payload* del respectivo mensaje a través del protocolo RF.

Este diseño modular podría permitir añadir interfaces que implementen un protocolo de envío de bytes *raw*, que es uno de los objetivos marcados.

El RF Discovery es otro módulo que descubre y enumera los NFC Endpoints, para cada uno devuelve información al DH, como el protocolo usado por la tarjeta para elegir la interfaz adecuada. Esta información es la que posteriormente obtiene el usuario en forma de objeto `Tag` a través del *Intent*.

NCI también incluye un módulo llamado *Listen Mode Routing* que permite al NFCC definir donde enrutar los datos recibidos. Para ello el DH mantiene una tabla de enrutamiento en el NFCC en base a la tecnología o protocolo del tráfico entrante, que enviará cada paquete al canal lógico correspondiente.

Durante la inicialización, la especificación dice que el NFCC debe enviar al DH la versión, módulos e interfaces soportadas, reglas de enrutamiento, etc., y podrá ser el DH el que se encargue, parcialmente, de configurar dichos parámetros. En el Apéndice B se añaden las tablas con algunos mensajes de configuración. Para mayor detalle se recomienda consultar la especificación NCI del NFC Forum [For].

En cualquier caso, la posibilidad de poder configurar parte del comportamiento del NFCC desde el propio DH, como la tabla de enrutamiento o las tecnologías y modos soportados, puede permitir ampliar las capacidades NFC sin necesidad de modificar el *firmware* o el sistema: tan sólo haría falta una aplicación nativa con permisos de superusuario que envíe mensajes NCI, lo que lo convierte en una línea de investigación muy interesante.

Una vez repasados la arquitectura y componentes principales de NCI, se puede encontrar en el Apéndice C una traza desde la llamada `transceive` de la API en una aplicación normal, hasta el envío del mensaje NCI a través del *driver* al NFCC. Allí queda patente que el *payload* original es encapsulado en distintos contenedores hasta llegar al NFCC, pero nunca llega a ser modificado. Para conseguir esta traza hizo falta realizar algunas modificaciones en el terminal que se encuentran documentadas en el Apéndice D.

3.3. Limitaciones: conclusiones

Una vez revisados los protocolos e implementación en Android, se puede concluir qué se puede hacer y qué no.

La primera limitación es que no es posible enviar órdenes directamente sobre ISO 14443-3. Como se ha visto, es el NFC Controller el que añade el *payload* al canal RF en función de la tecnología usada, lo que significa que el sistema operativo Android no tiene control sobre el CRC u otros campos. Sin embargo, si se llega a modificar el *firmware* posiblemente podría añadirse una interfaz que permita el envío de órdenes *raw*.

Del mismo modo, aunque el NFC Controller puede llegar a ponerse en modo escucha para diferentes tecnologías, la implementación HCE de Android sólo permite recibir a nivel software APDUs ISO 7816-4. Además se añade la restricción de que estos sólo serán enrutados si se ha realizado antes un `SELECT` explícito (cosa que no siempre hacen los

lectores).

Otra limitación a tener en cuenta es el *timeout* máximo permitido ya que cualquier *relay* va a incluir un retraso en la comunicación, así que dependiendo de la ventana de tiempo será posible utilizar una comunicación con menor o mayor latencia, que puede traducirse en una mayor o menor distancia. En la mayoría de trabajos mencionados en el Capítulo 5 se aborda este punto, y aunque los tiempos en ISO 14443-3A (parte de anti-colisión y selección) son bastante restrictivos, a nivel ISO 14443-4A la ventana de tiempo puede llegar hasta los 5 segundos, tiempo más que suficiente para cualquier canal de *relay* (ya sea sobre 3G, Bluetooth o WiFi). Para más detalle se remite al trabajo de [KH14], donde realizan un análisis exhaustivo sobre coste de tiempo y *timeouts* en distintos canales durante un ataque de *relay* NFC.

A pesar de esto, los dispositivos Android con soporte NFC y versión mayor que 4.4 brindan las suficientes características como para llevar a cabo un ataque de *relay* a nivel de APDUs (sobre ISO 14443-4). Para implementar el Proxy es necesario HCE (API Level 19) para emular la tarjeta, pero la Mole tan sólo hará uso de la clase `Tag` que se encuentra disponible a partir de la API Level 10. Los conceptos de Proxy y Mole se definen en detalle en el siguiente capítulo.

Capítulo 4

NFC Leech: diseño e implementación

El objetivo final de este proyecto es implementar un ataque de *relay* con una aplicación Android en un dispositivo sin modificar. En el capítulo anterior se han descrito las limitaciones para llevar a cabo dicho ataque. En este capítulo se van describir el diseño e implementación seguidos para desarrollar la aplicación NFC Leech que permitirá realizar este ataque de *relay*.

En la fase de análisis del problema se fijó primero la restricción de utilizar un dispositivo lo más genérico posible, de este modo la aplicación podrá ser accesible a usuarios de cualquier perfil técnico, lo que pone de relieve la gravedad: **ya no es necesario utilizar hardware especializado o modificar el software de un dispositivo para realizar un ataque de este tipo**. Por otro lado, se requiere una aplicación lo más modular posible siendo capaz de utilizar distintos canales de comunicación y que además pueda ser fácilmente integrada con cualquier otro dispositivo, por ejemplo, en un ordenador con lector de tarjetas NFC.

Además se quiere que el usuario pueda visualizar todo el proceso y ver los mensajes enviados y recibidos durante la comunicación. La aplicación pretende servir como prueba de concepto para mostrar cómo se puede llevar a cabo un ataque de *relay* real a una tarjeta de crédito *contactless*. El código no va a hacerse público, pero se facilitará a cualquier persona que quiera estudiarlo, siempre que sea con fines académicos o de investigación.

4.1. Diseño

Para realizar un buen diseño que cumpla con las características deseadas de modularidad e independencia se han utilizado los patrones de diseño: *Facade* y *Adapter*. El primero permite gestionar y reducir la complejidad de clases y la estructura orientada a *callbacks* de las APIs de Android, mediante la división en subsistemas. El segundo sirve para definir interfaces que serán implementadas por los dispositivos específicos o por los distintos canales, solucionando así los problemas de dependencia.

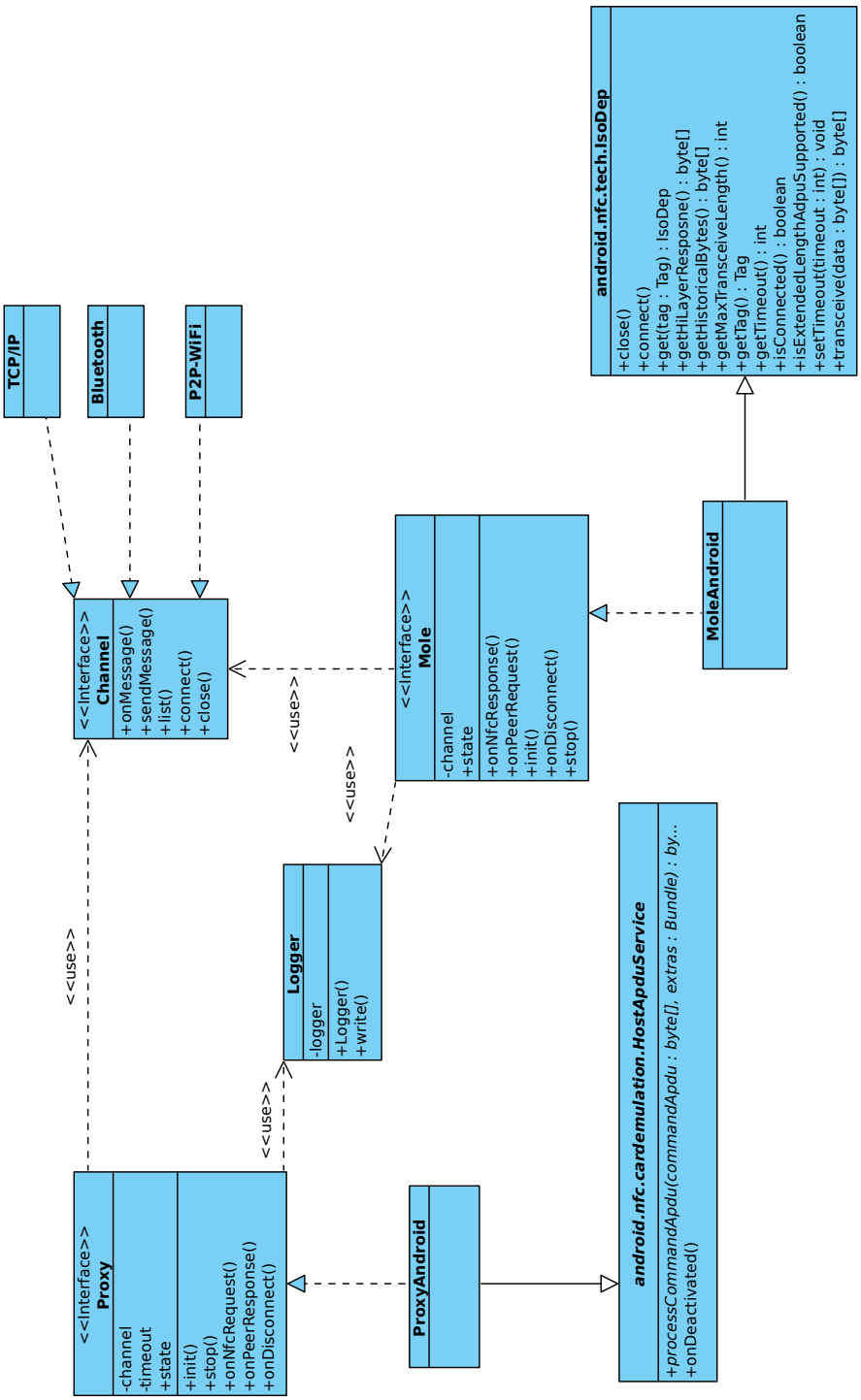


Figura 4.1: Diagrama de clases general de la aplicación NFC Leech.

A continuación, apoyándose en la Figura 4.1, se especifican los elementos necesarios para la implementación de la aplicación, así como su uso y la interacción con el usuario y entre ellos.

Para llevar a cabo el ataque, a nivel de hardware son necesarios dos dispositivos: uno que actúe como Proxy y otro como Mole, cuyos roles se explican más adelante. La aplicación implementa ambas funcionalidades y permite al usuario especificar el modo de funcionamiento en cada ejecución. También permite seleccionar el canal de comunicación que debe usarse entre: WiFi-Direct, TCP/IP y Bluetooth, utilizando por defecto WiFi-Direct. Esto último se ha conseguido utilizando el Adapter y un patrón Factory para la ejecución de actividades, donde Android lanzará una u otra en función de la configuración definida, resultando trivial la inclusión de otra interfaz de usuario para un canal nuevo.

Una vez seleccionados el modo de trabajo y el canal de comunicación, se establece la conexión entre ambas partes mediante *sockets*. A partir de este punto cada terminal trabaja según su rol:

Mole El par que actúa como **Mole** permanecerá a la escucha en el canal de *relay* hasta que llegue una solicitud, en ese momento se comunicará con la tarjeta retransmitiendo la orden y enviará la respuesta de la tarjeta, de vuelta al canal de *relay*.

Proxy Por otro lado el par que trabaje como **Proxy** se mantendrá a la espera de una solicitud por parte del PCD, al recibirla la retransmitirá por el canal de *relay* (momento en el que el PCD empieza a consumir su *timeout*). Una vez llega la respuesta por el canal de *relay* la retransmitirá inmediatamente al PCD, concluyendo así el proceso.

En la Figura 4.2 se puede ver la arquitectura de un ataque de *relay* NFC a nivel ISO 7816-4. En la que se ve el proceso descrito anteriormente donde PCD y Proxy establecen un canal lógico sobre NFC (ISO 14443-4), al igual que Mole y PICC, y donde Proxy y Mole se mantienen conectados por otro canal (el de *relay*) para retransmitir los mensajes.

Además, para que el usuario pueda ver todo el proceso en su terminal, cada evento debe generar un registro que sea almacenado y mostrado por pantalla. Dicho registro contiene el tipo de evento, su estado y en caso de ser un mensaje el *payload* (como su representación hexadecimal). Una de las propuestas de trabajo futuro es realizar un procesado de este contenido según la norma ISO/IEC 7816-4 y el estándar EMV, para mostrar mensajes más legibles y servir en los procesos de depuración de protocolos.

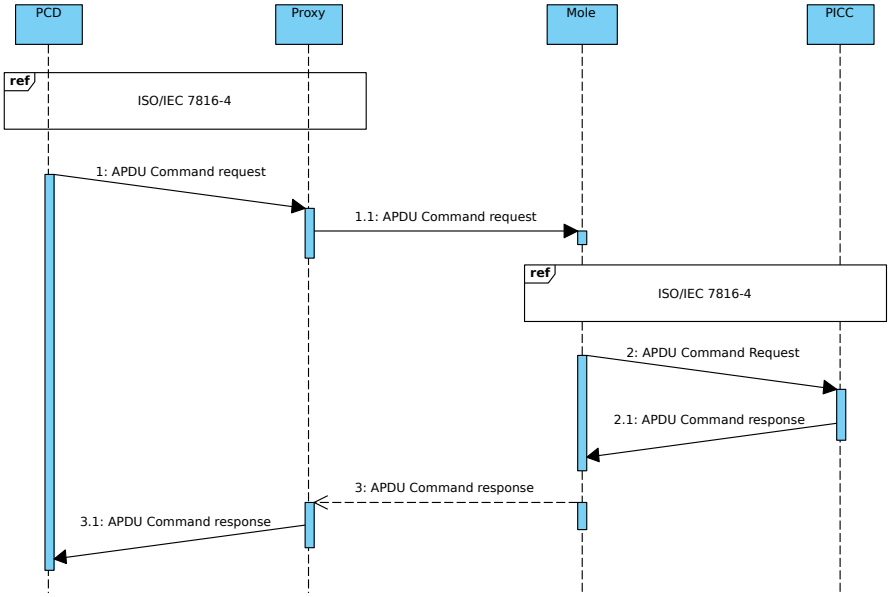


Figura 4.2: Arquitectura de *relay* de APDUs en NFC.

4.2. Implementación

4.2.1. Proxy

El Proxy es la pieza clave del escenario y como se ha visto es el encargado de utilizar HCE para trabajar en modo pasivo. Android lo implementa con un servicio que extiende la clase `HostApduService`, como se ha visto en el Capítulo 3. Además se ha optado por utilizar un servicio genérico `RelayService` corriendo sobre un *thread* distinto al principal para evitar congelar la interfaz de usuario. La principal razón es que el comportamiento de Proxy y Mole es bastante similar en cuanto a escucha de canales y retransmisión de información aunque usen componentes distintos. En la Figura 4.3 se puede ver un diagrama de la solución descrita.

Desafortunadamente Java no soporta herencia múltiple, por lo que la implementación se complica un poco. El servicio que extiende a `HostApduService` debe registrarse en el Manifest de la aplicación junto con los AID a los que quiere responder, y será ejecutado ofreciendo su interfaz a través del método `onBind` por el servicio NFC del sistema. La solución utilizada, ha sido crear el servicio `ProxyService` que corre en *background* durante todo el *relay* extendiendo la clase `RelayService` e implementando el *listener*, permitiendo la comunicación mediante mensajes con el servicio HCE gracias a la capacidad de *reflection* de Java.

Del mismo modo, el servicio se encargará de enviar los registros de cada evento a la actividad principal encargada de su gestión. Se ha optado por diferenciar entre 2 tipos de eventos:

- **Comandos;** son órdenes enviadas desde la actividad o el usuario para iniciar o

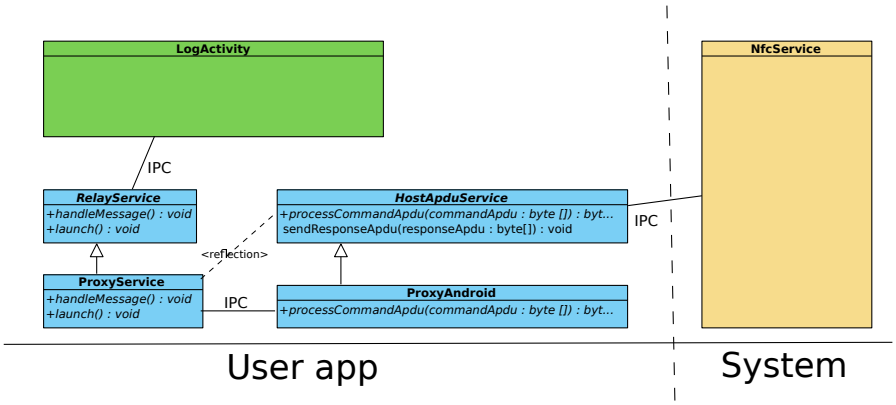


Figura 4.3: Diagrama del elemento Proxy.

detener el servicio.

- **Mensajes;** llevan el tipo de información y datos llegado a través de uno de los canales de comunicación. En el Proxy los mensajes podrán ser solicitudes del PCD que nos llegan vía NFC o respuestas del *tag* que nos llegan por el canal de *relay*.

Otra parte importante para que funcione es registrar correctamente los AID que usan las tarjetas bancarias, que dependen de la tarjeta y del fabricante, como se vio en el Capítulo 2. Brevemente, en el caso particular de los TPV, en lugar de almacenar una lista exhaustiva de AIDs[1], utilizan uno para una aplicación especial llamada PPSE (Proximity Payment Systems Environment[2]), que no es más que una interfaz que deben implementar las tarjetas de crédito que devuelve la lista de AIDs de la aplicaciones instaladas junto con su prioridad. De este modo, el TPV va a enviar normalmente dos órdenes `SELECT` antes de iniciar la transacción, el primero al PPSE y el segundo a la aplicación concreta que tenga registrada la tarjeta objetivo.

Por este motivo la aplicación desarrollada, NFC Leech, registrará el identificador del PPSE junto con todas las aplicaciones de las tarjetas que quiera soportar.

4.2.2. Mole

La Mole es la parte encargada de comunicarse con la tarjeta legítima, es decir, enviar al *tag* las órdenes que le llegan vía *relay* y retransmitir las respuestas de vuelta.

La implementación es muy parecida al Proxy; el servicio `MoleService` extiende a la clase `RelayService`, por lo que se trata de nuevo de un servicio que se ejecuta en *background* en un hilo independiente y que se comunicará mediante paso de mensajes con la actividad de gestión de eventos.

En este caso es necesario utilizar la clase `IsoDep` que implementa la tecnología ISO 14443-4 y, como se vio en el Capítulo 3, está basada en `BasicTechnology`. Esta cla-

[1]Lista de AIDs: https://en.wikipedia.org/wiki/EMV#Application_selection
[2]AID PPSE: 325041592E5359532E4444463031

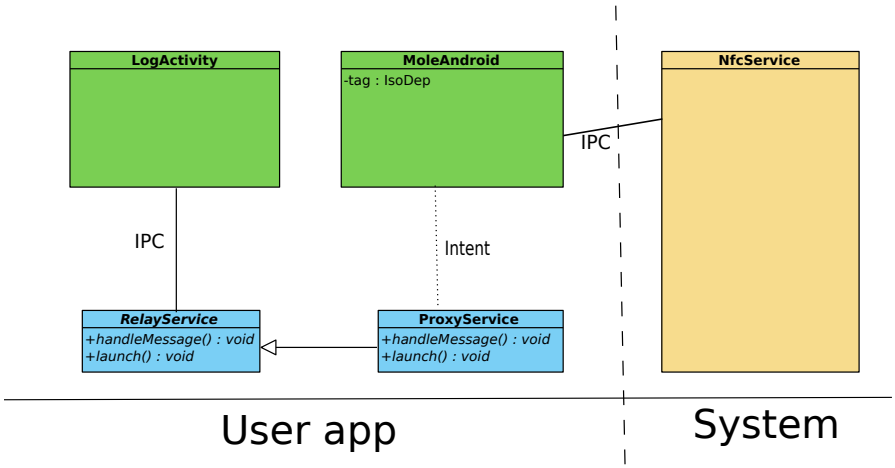


Figura 4.4: Diagrama del elemento Mole.

se ofrece un método `transceive` que permite enviar una orden APDU y devuelve la respuesta del *tag*, o lanza una excepción en caso de error o *timeout*.

En la Figura 4.4 se describen, igual que en la parte anterior, las clases participantes y su interacción.

De nuevo surge un problema en la implementación en Android y es que el sistema no permite registrar un servicio para recibir los *Intents* con el `Tag`, tan sólo pueden hacerlo aplicaciones corriendo en *foreground*. Para solucionarlo se ha implementado una actividad invisible, `MoleAndroid`, que al recibir el *Intent* simplemente lo reenvía al servicio mediante un mensaje de *broadcast* y finaliza.

4.2.3. Canal de retransmisión

El canal de *relay* o de retransmisión es el encargado de enlazar a Proxy y Mole. Para ello se ha implementado una clase que establece un *socket* y permite a los pares trabajar tanto de cliente como de servidor, independientemente de su rol. El `RelayService` será el encargado de iniciar dicha conexión y gestionará los eventos del canal (nuevo mensaje o fin de comunicación).

Se ha puesto especial énfasis en hacer esta parte de comunicación independiente del medio utilizado, para ello se ha diseñado una interfaz `Channel`, siguiendo el patrón Adapter, que deberá ser implementada por los distintos canales.

El usuario podrá elegir en su configuración a través del menú de opciones de la aplicación principal, qué tipo de canal utilizar y, dependiendo de la selección, al iniciar el proceso se lanzará una actividad u otra gracias al uso del patrón Factory. La configuración define WiFi-Direct como canal de comunicación *relay* por defecto.

WiFi-Direct WiFi-Direct es un protocolo que permite a varios dispositivos WiFi conectarse entre sí sin necesidad de un punto de acceso. Las principales ventajas de usar

este canal son dos:

- La primera es que permite establecer una conexión entre dos dispositivos de manera rápida y sin necesidad de configuración, lo que puede resultar crítico para llevar a cabo un ataque en un momento puntual.
- La segunda es que Android ofrece una API[3] Level 14 basada en *callbacks* bien documentada y fácil de utilizar, una vez se entiende su funcionamiento global.

Las desventajas:

- Al utilizar WiFi la distancia entre los dispositivos tiene un límite y suele ser alrededor de los 50 metros.
- Android ofrece una API basada en *callbacks*, y puede ser bastante complicado integrarla en una aplicación si no se ha trabajado antes con este paradigma.

La implementación ofrece al usuario una interfaz simple donde se listan los dispositivos cercanos y permite establecer una conexión mediante un simple clic (véase captura derecha de la Figura 4.10).

TCP/IP En este modo se permite al usuario utilizar un canal ya establecido por el sistema y simplemente introducir una dirección IP y un puerto al que conectarse. Este método permite realizar una comunicación a través de 3G o de una conexión WiFi lo que significa que Proxy y Mole pueden estar separados varios kilómetros mientras el canal tenga una latencia baja. En [KH14] se presentan datos y un análisis detallados de distintas latencias en diferentes canales en relación a los límites que se describen a continuación.

En la ecuación 4.1 se definen las fórmulas del *Frame Delay Time* (FDT), que es el tiempo entre dos *frames* durante las fases de inicialización y anti-colisión. El FDT máximo es 86.4 μs.

$$FDT = \frac{n \cdot 128 + 20}{f_c} \tag{4.1}$$

Durante la configuración del canal se calcula el *Frame Waiting Time* (FWT), que será el tiempo de espera del PCD para una respuesta. Está descrito en la ecuación 4.2 a partir del campo FWI que se define mediante la orden `ATS`, como se vio en la Sección 2.1. El estándar permite valores entre 302 μs y 4989 ms, un margen muy grande para el *relay*. Además se define un *Waiting Time Extension* (o WTX) para que el PICC solicite tiempo extra al PCD durante operaciones computacionalmente costosas, de este modo el *timeout* puede extenderse ilimitadamente.

$$FWT = 256 \cdot (\frac{16}{f_c}) \cdot 2^{FWI}, 0 \leq FWI \leq 14 \tag{4.2}$$

[3] https://developer.android.com/guide/topics/connectivity/wifip2p.html

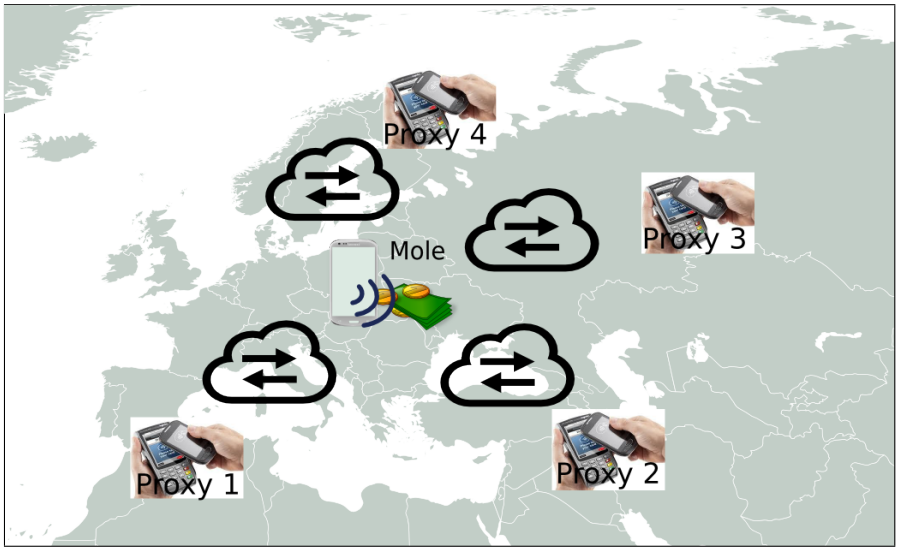


Figura 4.5: Ejemplo de ataque de *relay* NFC; una tarjeta a N sitios.

También permite conectar de manera sencilla la aplicación móvil con cualquier otro dispositivo conectado a Internet, y gracias a la interfaz *Channel* se puede implementar un *wrapper* en Java para enviar y recibir mensajes utilizando el hardware NFC de un dispositivo cualquiera. La desventaja es que requiere una IP pública y más tiempo de configuración, pero dependiendo del escenario será una solución adecuada.

Un ejemplo puede ser el caso de dejar una tarjeta cerca de un dispositivo fijo que actúe como Mole y crear un canal de *relay* desde un móvil (Proxy) a éste, así **la tarjeta podría ser usada por distintas personas desde distintos lugares a la vez**. Esto podría usarse para, entre muchas otras cosas, hacer realmente complicado el rastreo de una tarjeta. En la Figura 4.5 se describe este escenario de ataque.

El caso contrario también puede resultar interesante si se imagina, por ejemplo, un software malicioso que convierte los dispositivos infectados en Proxies NFC. El criminal podría **utilizar un dispositivo como Mole y su *botnet* para pagar en casi cualquier momento** ya que las posibilidades de que al menos un dispositivo infectado este cerca de una tarjeta en un momento dado son bastante altas. En la Figura 4.6 se puede ver el ejemplo de este escenario de ataque.

Bluetooth Este canal no se ha llegado a implementar, pero se deja para mejoras futuras. La idea es utilizar otro medio como Bluetooth que de nuevo permite una configuración rápida y sencilla para dispositivos cercanos (hasta 30 metros), y que además tiene buen soporte por parte de Android. Especialmente a partir de Android 4.3 (API Level 18), que al dar soporte a BLE (*Bluetooth Low Energy*) puede suponer una mejora considerable en el consumo de batería, lo que resulta un aspecto crítico en dispositivos móviles.

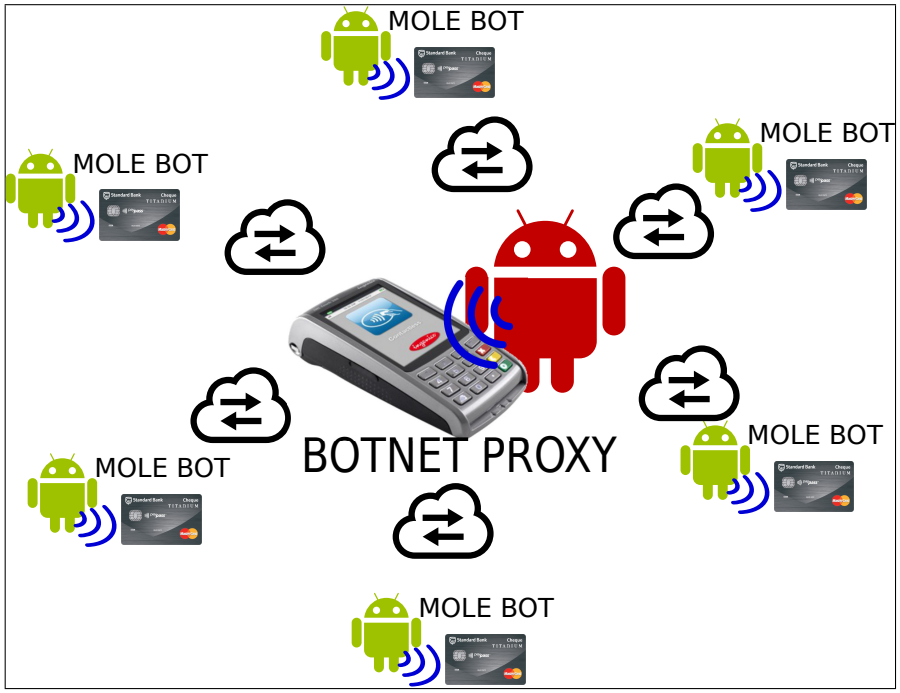


Figura 4.6: Ejemplo de ataque de *relay* NFC; *botnet* de tarjetas de pago.

4.3. Prueba de concepto

La aplicación llevada a cabo ha resultado de una complejidad alta ya que ha sido necesaria la interacción entre distintos módulos y APIs, por lo que el autor ha tenido que aprender a programar en Android. Al final se ha implementado la aplicación llamada NFC Leech, capaz de realizar un ataque de relay NFC con dispositivos Android en un escenario de cobro con una tarjeta de crédito *contactless*. Para llevar a cabo las pruebas se han utilizado:

- Dos dispositivos móvil Android (véase la Figura 4.7) con soporte NFC (uno de ellos con versión $\geq$4.4).
- Un TPV (Terminal Punto de Venta) Ingenico IWL8280, que se muestra en la Figura 4.8, con soporte *contactless* (la mayoría de comercios los utilizan ya), proporcionando la capacidad de realizar pruebas de pago reales en un entorno controlado.
- Una tarjeta de crédito MasterCard del Banco Sabadell (las tarjeta nuevas de Banco Sabadell, BBVA, Bankia y Santander ya soportan NFC; al lado del chip hay un dibujo de ondas de radiofrecuencia, como se ve en la Figura 4.9), **que permite el pago sin contacto para transacciones menores de 20 euros, sin solicitar el código de seguridad o PIN**.

Figura 4.7: Dispositivo Nexus 4; con Android 4.4 y soporte NFC.

Figura 4.8: TPV Ingenico IWL280; GPRS y NFC.

Figura 4.9: Tarjeta de crédito NFC del Banco Sabadell.

Se ha montado un escenario en el que entran en juego los 4 agentes descritos. En primer lugar situaremos a la Víctima, que dispone de una tarjeta de crédito *contactless* en su cartera. Por otro lado tendremos el Vendedor que quiere cobrar un servicio o producto de menos de 20 euros de coste, y por último a los atacantes A y B. El Atacante A ejecutará su aplicación en modo Proxy mientras que el B en modo Mole, y una vez establecida la comunicación el primero se dirigirá a pagar a la vez que el Atacante B situará su terminal cerca de la cartera de la Víctima. En el momento que el Atacante A acerque su dispositivo móvil al terminal de venta el pago se completará a cargo de la Víctima, sin que ninguna parte se de cuenta. En las Figuras 4.10 y 4.11 se muestran una captura de pantalla de la aplicación y otra del vídeo de demostración donde se representa el ataque: a la izquierda se ve a la Mole conectada a una tarjeta y a su derecha el Proxy en proximidad con el TPV, haciendo el *relay*.

Hay que tener en cuenta que los bancos están apostando por sistemas de pago vía móvil[4] como ya se lleva haciendo en otros países algunos años, y no tardará en empezar aplicarse a gran escala, por lo que estos escenarios se harán cotidianos y otros nuevos aparecerán en los próximos años.

[4]http://bspress.bancsabadell.com/2014/02/mastercard-host-card-emulation-pagos-moviles-nfc.html

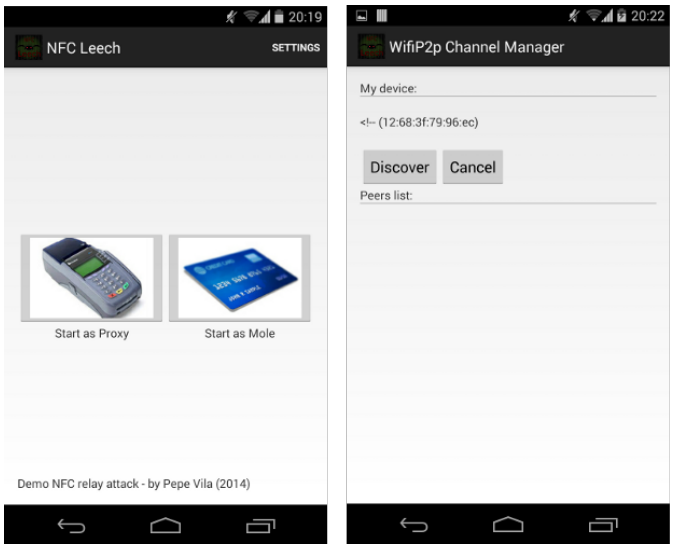


Figura 4.10: Capturas de pantalla de la aplicación: pantalla principal a la izquierda, y selección de pares mediante WiFi-Direct a la derecha.

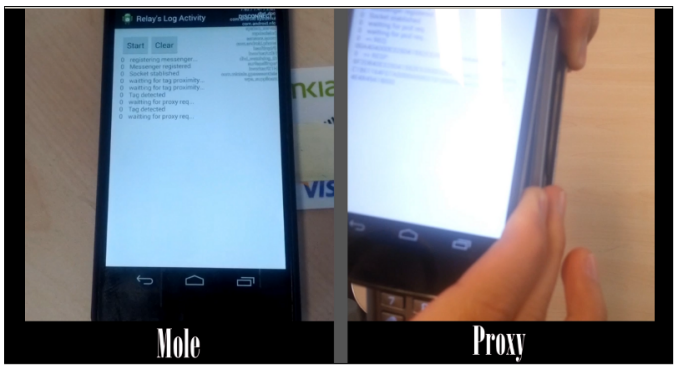


Figura 4.11: Captura de pantalla de vídeo demo realizando un ataque de relay con NFC Leech.

Capítulo 5

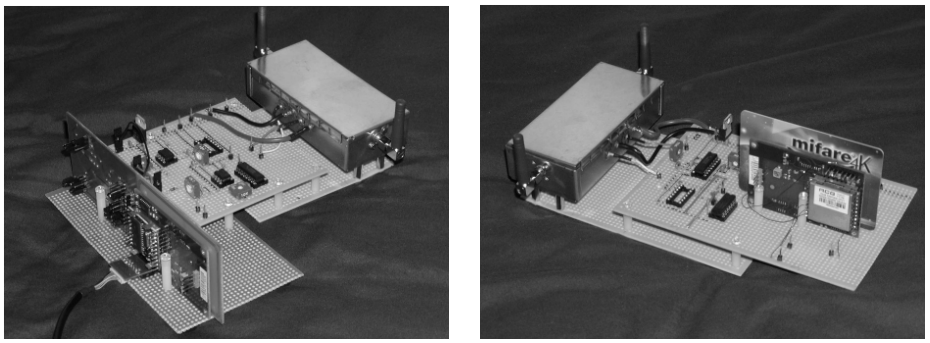
Trabajo relacionado

En este capítulo se pretende llevar a cabo un repaso cronológico y exhaustivo de todos los trabajos previos sobre ataques de *relay* NFC, con el fin de ver los avances producidos y las limitaciones todavía existentes.

Febrero de 2005 En [Han05] se presenta el primer ataque práctico de *relay* contra tarjetas RFID ISO 14443-3A. Para ello utiliza dos elementos que se ven en la Figura 5.1: el *Proxy* y la *Mole*, donde el primero actúa como una tarjeta válida y retransmite las instrucciones a la Mole que responde tras comunicarse con la tarjeta legítima.

Este trabajo utiliza hardware *ad hoc* y para conseguir bajas latencias y un menor *delay* colocan un modulador/demodulador en cada extremo para realizar un *relay* de cada paquete usando radio frecuencia (alcance de 50m). La principal desventaja de esta solución, aunque más rápida y sencilla de implementar, es que no permite inspeccionar el tráfico ni modificarlo, ya que no se llegan a reconstruir las órdenes.

Aunque la aplicación desarrollada en este PFC, NFC Leech, no logra trabajar a nivel ISO 14443-3A, sí que permite inspeccionar el tráfico interceptado y no



(a) Proxy (b) Mole

Figura 5.1: Hardware *ad-hoc* para implementar ataque de *relay* NFC.

Method	Max Distance	Extra Cost (beyond NFC)	Availability	Attacker Knowledge
Standard	10 cm	0$	Hight	Low
Current + Antena	40 cm	≤ 100	Hight	Medium
Current + Anteana + Software	50 cm	≥ 100	Medium	Low
Current + Antena + Signal-Processing	55 cm	≥ 5000	Low	Very Hight

Tabla 5.1: Tabla de costes/propiedades extraída de [KW05].

requiere hardware propio, lo que la convierte en un candidato más práctico para un ataque real.

Al final en [Han05] queda claro que las restricciones de tiempo no son suficientes para hacer inviable el ataque, y que **la seguridad de la capa de aplicación es inútil ante un *relay*, por lo que nuevas medidas deberían establecerse a nivel físico**.

Septiembre de 2005 En [KW05] implementan un ataque parecido al anterior, pero centrando sus esfuerzos en mejorar la distancia a la que puede ser leída una tarjeta, alcanzando hasta los 50cm. La Tabla 5.1 mide el coste en hardware, software y conocimientos que debe invertir un atacante para lograr distintos beneficios.

De nuevo utilizan hardware propio muy especializado, y cambian de nomenclatura: llamando *Ghost* (o Fantasma) al *Proxy* y *Leech* (o Sanguijuela) a la *Mole*.

En este caso presentan un escenario en el que el atacante, por ejemplo, coloca el *Leech* cerca de la víctima para leer una tarjeta de crédito que lleve en la cartera. Proponen el uso de tarjeteros-jaulas de Faraday [How08], o tarjetas con pines de encendido que sólo funcionen previa interacción del usuario.

Este trabajo abre la puerta a nuevas variantes de software malicioso muy interesantes y que probablemente empiecen a aparecer muy pronto en el ecosistema. La idea consiste en infectar un dispositivo móvil y extraer la información de tarjetas cercanas utilizando sus características NFC para posteriormente transmitirla a los criminales. Escenarios de ataque posibles con software malicioso han sido descrito en más detalle en la Sección 4.2.3.

Junio de 2007 En [DM07] se implementa por primera vez un escenario de fraude en el sistema de pago EMV mediante ataques de *relay*. En este caso se obliga a la víctima a realizar un pago en un TPV (Terminal Punto de Venta) manipulado que actuará de Proxy para cobrarle una cantidad mayor en otra parte, de este modo se puede llevar el ataque a cabo con ventas mayores de 20 euros ya que el usuario introducirá su PIN en el terminal malicioso y éste será retransmitido también. De nuevo utiliza hardware a medida, aunque menciona la posibilidad de utilizar los móviles en un sistema de pago.

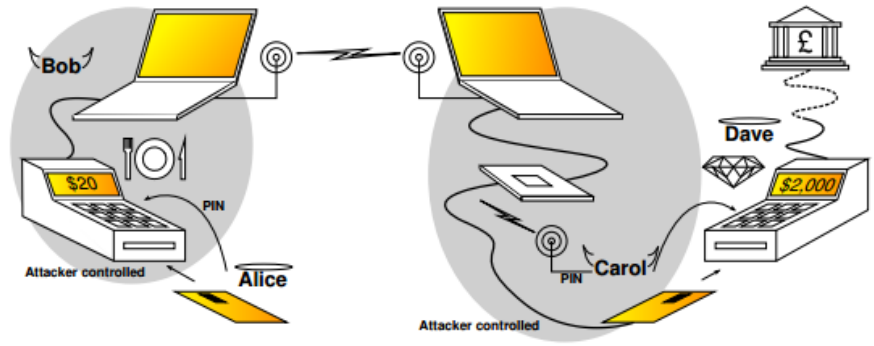


Figura 5.2: Arquitectura de ataque *relay* con un punto de venta malicioso.

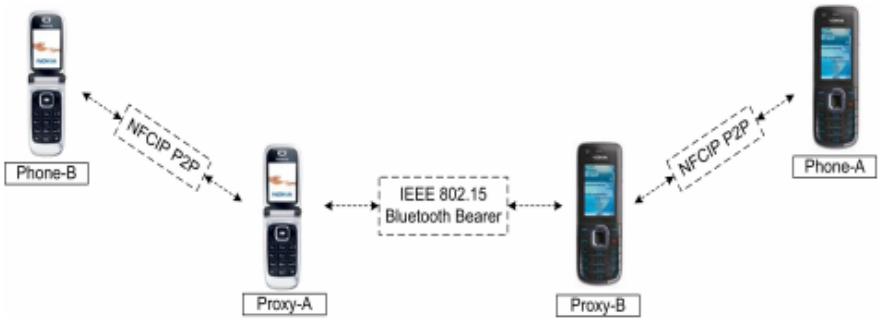


Figura 5.3: Arquitectura de ataque *relay* en comunicación P2P.

El escenario, que se muestra en la Figura 5.2, es distinto al planteado en este proyecto pero sirve como ejemplo de otros vectores de ataque.

Junio de 2010 Con el creciente interés de las comunicaciones *peer-to-peer* entre dispositivos móviles, se empieza a utilizar NFC en las aplicaciones para intercambiar archivos, establecer configuraciones o jugar. En [FHMM10] demuestran un ataque de *relay* a una comunicación NFC *peer-to-peer* utilizando, por primera vez, tan sólo móviles comerciales con una aplicación que usa la API del fabricante.

Para ello utilizan cuatro terminales Nokia:

1. Phone A: Nokia 6212 Classic NFC
2. Proxy A: Nokia 6131 NFC
3. Phone B: Nokia 6131 NFC
4. Proxy B: Nokia 6212 Classic NFC

En este escenario de ataque descrito en la Figura 5.3, se muestra que Proxy B va a representar a Phone B y retransmite la comunicación en el *relay* hacia y desde

Phone A (Proxy A realiza el proceso análogo). Todos son dispositivos basados en Symbian S40 y utilizan una aplicación Java usando las API del fabricante para comunicaciones NFC *peer-to-peer* y Bluetooth (para el canal de *relay*).

El experimento les permite hacer un *relay* completo de una comunicación NFC P2P, y proponen utilizar un tercero de confianza para confirmar la localización de los dispositivos evitando así el ataque.

A partir de aquí se ve la tendencia de que a medida que los fabricantes hacen sus dispositivos más accesibles y liberan APIs para el desarrollo de aplicaciones NFC, los móviles se convierten en herramientas de ataque tan potentes como cualquier dispositivo especializado.

De nuevo el objetivo es distinto del propuesto en este trabajo, pero podría considerarse otro caso de estudio el trasladar un ataque de *relay* NFC P2P a Android, ya que todavía no se ha hecho.

Mayo de 2010 En [Wei10] se presenta uno de los trabajos más completos en seguridad NFC y ataques de *relay*. En él se describen distintas arquitecturas de ataques de *relay* y sus limitaciones, para finalmente implementar un ataque de *relay* con un dispositivo Nokia 6121 Classic NFC como Mole, y una placa Beagle Board con Android conectada a un USB NFC (compatible con *libnfc*[1], que sí permitir enviar órdenes ISO 14443-3A *raw*) como Proxy.

Sin embargo, el *relay* queda restringido a ISO 14443-4, debido a las limitaciones de tiempo de ISO 14443-3A y a que la API de Nokia sólo soporta trabajar con APDUs. En cualquier caso el trabajo realizado por Weiss es sobresaliente además de pionero, aunque lamentablemente todavía no resulta posible utilizar tan sólo dispositivos móviles.

Febrero de 2012 En [FHM+12] presenta el primer ataque de *relay* estándar utilizando tan sólo dispositivos móviles, como ya se había vaticinado. Debido a la explosión de dispositivos móviles con soporte NFC como el Nokia C7, RIM Blackberry 9900/9930 o el Google Nexus S, los fabricantes apuestan por dar más libertad a los desarrolladores de aplicaciones con el soporte a emulación de *tags* vía software. El primer ejemplo es la Blackberry 9900, aunque también es posible utilizar un Google Nexus S con un *firmware* modificado[2]. Además empiezan a aparecer sistemas de pago NFC con dispositivos móviles como Google Wallet u Orange QuickTap.

Este ataque (véase la Figura 5.4), consigue realizar un *relay* a nivel de APDUs igual que en el trabajo que aquí se presenta, pero se trata de dispositivos concretos con una menor cuota de mercado y, sobretodo, con una dificultad alta para la instalación de aplicaciones, con lo que todavía resulta poco práctico.

Junio de 2012 Douge Yeager realiza dos commits[3,4] en CyanogenMod 9, consiguiendo

[1] https://code.google.com/p/libnfc/
[2] http://www.nfcworld.com/2011/02/13/35913/
[3] https://github.com/CyanogenMod/android_frameworks_base/commit/c80c15bed5b5edffb61eb543e31f0b90eddcdadf
[4] https://github.com/CyanogenMod/android_external_libnfc-nxp/commit/34f13082c2e78d1770e98b4ed61f446beeb03d8

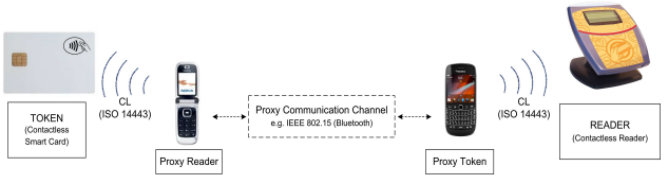


Figura 5.4: Ataque *relay* NFC con Nokia y Blackberry.

así soporte para emulación software de tarjetas en dispositivos Android (con chip de NXP semiconductors). A raíz de este trabajo, Eddie Lee presenta en DEFCON 20 [Lee12] la herramienta NFC Proxy, que permite a cualquier persona con un dispositivo móvil NFC Android y una versión de CyanogenMod +10 instalada realizar ataques de *relay* de APDUs.

En este caso el impacto es mayor y queda clara la aplicación práctica y la facilidad con que se puede implementar el ataque, pero sigue siendo necesario utilizar una ROM modificada, lo que restringe la distribución a diferencia de NFC Leech.

Octubre de 2013 Desde la modificación de CyanogenMod han hecho falta casi dos años hasta que Android diera soporte nativo a la emulación por software con el lanzamiento de la versión 4.4 (KitKat)[5]. HCE (Host Card Emulation) permite a los desarrolladores programar aplicaciones móviles que actúen como elemento pasivo a nivel de ISO 14443-4 (IsoDep), emulando tarjetas inteligentes que sigan el estándar ISO/IEC 7816-4.

Sin embargo, HCE sigue teniendo dos limitaciones:

- Requiere que el lector envíe una orden `SELECT AID` explícito para seleccionar la aplicación de la tarjeta con la que se quiere comunicar.
- La aplicación Android debe registrar en su Manifest la lista de AIDs a los que responder.

Diciembre de 2013 Johannes Zweng publica en el foro XdaDevelopers un módulo [Zwe13] para Xposed que permite redirigir todos los APDUs (u órdenes ISO/IEC 7816-4) entrantes a una aplicación concreta. Aunque para usarlo es necesario tener un dispositivo *rooteado* y el *framework* instalado.

A raíz de esto en abril de 2014 aparece en el Play Store una aplicación llamada NFCSpy[6] que además de utilizar opcionalmente este módulo, implementa un *relay* NFC para observar la comunicación entre una tarjeta y un terminal. Ésta es la primera demostración práctica de un ataque de *relay* NFC utilizando sólo dispositivos Android de serie y su impacto es crítico, ya que cualquier usuario con un móvil Android y soporte NFC puede instalar la aplicación con un simple clic.

[5]https://developer.android.com/about/versions/kitkat.html
[6]https://play.google.com/store/apps/details?id=com.sinpo.nfcspy

NFC Leech implementa el mismo ataque como prueba de concepto, utilizando un diseño más genérico que permita, además de distintos canales de comunicación, una fácil integración con otros dispositivos para poder llevar a cabo ataques más elaborados.

Sin embargo a día de hoy todavía no se ha conseguido llevar a cabo un ataque de *relay* a nivel ISO 14443-3A con dispositivo Android por defecto. Este trabajo quiere servir de base para explorar las posibles vías que pueden llevar a dar este salto.

También cabe mencionar que la mayoría de trabajos describe mecanismos de defensa ante los ataques de *relay*, generalmente mediante protocolos de *distance bounding* que intentan medir la distancia real entre el lector y *tag* legítimo a través de otros canales o de terceros (como GPS).

En resumen, las principales ventajas de NFC Leech respecto a los trabajos anteriores son:

1. Utiliza dispositivos móviles sin necesidad de hardware adicional.
2. Al ser una aplicación Android que utiliza la API pública, es compatible con cualquier dispositivo comercial que soporte NFC y su instalación y distribución pueden ser extremadamente sencillas gracias a los *markets*.
3. Gracias a su diseño puede ser fácilmente integrada con otras plataformas para construir escenarios de ataque más elaborados.

Por contra, la mayor desventaja es que el *relay* NFC con un dispositivo Android sigue restringido a la tecnología IsoDep, lo que limita su uso en tarjetas que sigan protocolos propietarios.

Capítulo 6

Conclusión y líneas futuras

En este último capítulo se presentan los resultados y conclusiones del proyecto, así como sus implicaciones. Además, se plantean posibles líneas futuras de trabajo.

A lo largo de este trabajo se ha realizado un análisis de la implementación NFC en Android obteniendo sus posibilidades y limitaciones para llevar a cabo ataques de *relay* o retransmisión sobre NFC. Además se ha conseguido implementar exitosamente la aplicación NFC Leech haciendo posible realizar el ataque en un escenario real de pago con tarjetas de crédito.

La tendencia actual parece indicar que las aplicaciones móviles y el pago con éstas vía NFC va a crecer en los próximos años y, al igual que con sus predecesores, se van a convertir en un objetivo para las mafias del fraude y la industria del software malicioso. Habiéndose demostrado que no hacen falta grandes recursos ni hardware especializado para llevar a cabo un ataque, el problema resulta todavía más crítico. De hecho, puede que en los próximos meses algunos de los escenarios de ataque que se han presentado aquí acaben convirtiéndose en realidad. Por este motivo hay que insistir y recordar que los nuevos sistemas o tecnologías no pueden sacrificar su seguridad, aunque como en este caso aporten mayor comodidad a los usuarios para realizar pequeños pagos.

Con este proyecto se pretende poner de relieve el problema de las tarjetas NFC y, en la medida de lo posible, alertar a usuarios y consumidores para evitar posibles daños. Es necesario recordar que a pesar de que los fabricantes pueden implantar medidas de seguridad a día de hoy, el problema radica en que éstos y las entidades bancarias ya han realizado una inversión y hasta que no obtengan un retorno, seguramente se tenga que convivir con este defecto.

6.1. Líneas de investigación futuras

Debido a la diversidad de temas estudiados y aprendidos para llevar a cabo este trabajo, en algunos puntos no se ha podido profundizar tanto como al autor le hubiera gustado. Por ese motivo en este apartado se enumeran las posibles líneas de investigación futuras que podría ser interesante llevar a cabo.

- **Posibilidades de configuración con NCI:** Investigar las posibilidades de configuración del NFCC a través de la NCI. De este modo podrían extenderse las capacidades del dispositivo móvil sin más necesidad que un dispositivo con permisos de super-usuario, aunque posiblemente haría falta realizar ingeniería inversa.
- **Custom *firmware*:** Modificar el *firmware* del NFCC. Esta solución es la más radical y sería completamente dependiente del hardware, pero realizar ingeniería inversa y modificar el binario sería la manera de conseguir mayor control sobre el dispositivo. El objetivo debería centrarse en la parte de interfaces que se explicó en el Capítulo 3 para conseguir que el dispositivo pueda enviar órdenes directamente sobre ISO 14443-3. Con esta solución se podría intentar emular protocolos propietarios y, tal vez, dar soporte a los dispositivos con chip BCM para tarjetas Mifare, que nativamente no están soportadas.
- **Librería Java para NFC *Relay*:** En el desarrollo de la aplicación se puso especial interés en un diseño modular, otra línea interesante sería mejorar dichas interfaces Java y ofrecer una pequeña librería que permita la integración de aplicaciones *relay* NFC independientemente del dispositivo (ya sean Androids con una ROM modificada, PCs, etc.).
- **Procesamiento de mensajes APDU:** Aunque la aplicación desarrollada muestra el contenido de los mensajes (órdenes y respuestas) entre el PCD y el *tag*, sería interesante llevar a cabo un procesamiento de los mensajes y mostrar al usuario información más legible que facilite, por ejemplo, la comprensión de órdenes EMV y la fácil depuración de aplicaciones NFC.
- **Análisis de seguridad de aplicaciones bancarias NFC:** Los bancos españoles están lanzando versiones de prueba de sus monederos digitales, con lo que sería interesante realizar un estudio comparativo de las distintas implementaciones y vulnerabilidades en éstas.
- **Vectores de explotación y software malicioso:** Inspección y análisis de muestras de software malicioso para dispositivos móviles, viendo su uso de las características NFC, así como la documentación de posibles ataques todavía no explotados.

Acrónimos

AID Application Identifier.

AOSP Android Open Source Project.

APDU Application Protocol Data Unit.

API Application Programming Interface.

ATM Automated Teller Machine.

ATQ Answer to Request.

ATS Answer To Select.

BLE Bluetooth Low Energy.

CRC Cyclic Redundancy Check.

DH Device Host.

EMV Europay, MasterCard y Visa.

HCE Host Card Emulation.

IEC International Electrotechnical Comission.

IPC Inter Process Communication.

ISO International Organization for Standardization.

JVM Java Virtual Machine.

NCI NFC Controller Interface.

NFC Near Field Communication.

NFCC NFC Controller.

NFCEE NFC Execution Environment.

P2P Peer-to-Peer.

PCD Proximity Coupling Device.

PFC Proyecto de Fin de Carrear.

PICC Proximity Integrated Circuit Card.

PIX Propietary Identifier Extension.

POS Point of Sale.

PPS Protocol and Parameter Selection.

PPSE Proximity Payment Systems Environment.

RATS Request ATS.

RF Radio Frequency.

RFID Radio Frequency IDentification.

RID Registered Application Provider Identifier.

ROM Read-Only Memory.

SDK Software Kit Development.

TPV Terminal Punto de Venta.

Bibliografía

[CON76] J. H. CONWAY. *On Numbers and Games*. Academic Press, 1976.

[DGB87] Yvo Desmedt, Claude Goutier, and Samy Bengio. Special Uses and Abuses of the Fiat-Shamir Passport Protocol. In *CRYPTO*, pages 21–39, 1987.

[DM07] Saar Drimer and Steven J. Murdoch. Keep Your Enemies Close: Distance Bounding Against Smartcard Relay Attacks. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 7:1–7:16, Berkeley, CA, USA, 2007. USENIX Association.

[FHM+12] Lishoy Francis, Gerhard Hancke, Keith Mayes, Konstantinos Markantonakis, and Information Security Group. Practical Relay Attack on Contactless Transactions by Using NFC Mobile Phones. IACR Cryptology ePrint Archive, 2012.

[FHMM10] Lishoy Francis, Gerhard Hancke, Keith Mayes, and Konstantinos Markantonakis. Practical NFC Peer-to-Peer Relay Attack Using Mobile Phones. In SiddikaBerna Ors Yalcin, editor, *Radio Frequency Identification: Security and Privacy Issues*, volume 6370 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin Heidelberg, 2010.

[For] NFC Forum. NFC Forum Technical Specifications. Available at `http://nfc-forum.org/our-work/specifications-and-application-documents/specifications/nfc-forum-technical-specifications/`.

[Goo14] Google. Android developers, 01 2014. `http://developer.android.com/reference/`.

[Han05] Gerhard Hancke. A practical relay attack on ISO 14443 proximity cards. Technical report, 2005.

[How08] HowToWired. *Make a Faraday Cage Wallet*, 2008.

[ISO10a] Identification cards – Contactless integrated circuit cards – Proximity cards. Part 2: Radio frequency power and signal interface, 2010. Available at `http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=39693`.

[ISO10b] Identification cards – Contactless integrated circuit cards – Proximity cards. Part 4: Transmission protocol, 2010. Available at http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50648.

[ISO11] Identification cards – Contactless integrated circuit cards – Proximity cards. Part 3: Initialization and anticollision, 2011. Available at http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=50942.

[ISO13] Identification cards – Contactless integrated circuit cards – Proximity cards. Part 1: Physical characteristics, 2013. Available at http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=39693.

[KH14] Thomas Korak and Michael Hutter. On the Power of Active Relay Attacks using Custom-Made Proxies. In IEEE, editor, *2014 IEEE International Conference on RFID (IEEE RFID) (IEEE RFID 2014)*, 2014. in press.

[KW05] Ziv Kfir and Avishai Wool. Picking Virtual Pockets using Relay Attacks on Contactless Smartcard Systems. *IACR Cryptology ePrint Archive*, 2005:52, 2005.

[Lee12] Eddie Lee. NFC Hacking: The easy way. 2012.

[Sem11] NXP Semiconductors. Mifare classic 1k - mainstream contactless smart card ic for fast and easy solution development, 2011.

[Wei10] Michael Weiß. Performing Relay Attacks on ISO 14443 Contactless Smart Cards using NFC Mobile Equipment. Master's thesis, Technische Universität München, May 2010.

[Zwe13] Johannes Zweng. Xposed Module NFC AID Re-Routing - https://github.com/johnzweng/XposedModifyAidRouting, October 2013.

Apéndice A

Horas de trabajo

Para el control del tiempo se utilizó una hoja de cálculo, se puede ver el tiempo dedicado en la Figura A.1 en forma de diagrama de Gantt.

La primera fase es el análisis y estudio del entorno en el campo de la seguridad RFID, necesario para poder decidir el alcance del proyecto. Una vez completo se llevó a cabo la parte más costosa: la fase de investigación y formación, donde se requerían una gran cantidad de conocimientos previos al tener que lidiar con un conjunto muy amplio de tecnologías y especificaciones. Una vez llevado a cabo el análisis y definidos los objetivos se aprendieron las nociones de programación en Android necesarias para desarrollar la aplicación NFC Leech. Los dos últimos meses se realizó el desarrollo de la aplicación y la elaboración de la memoria, junto con las pruebas en un escenario real con un TPV y una tarjeta de crédito *contactless*.

Al final de estos seis meses, se ha estimado un coste aproximado de 550 horas de trabajo entre las distintas fases de este proyecto. Lo que sobrepasa un poco la estimación inicial, pero que ha sido necesario dada la extensión de los conocimientos previos necesarios.

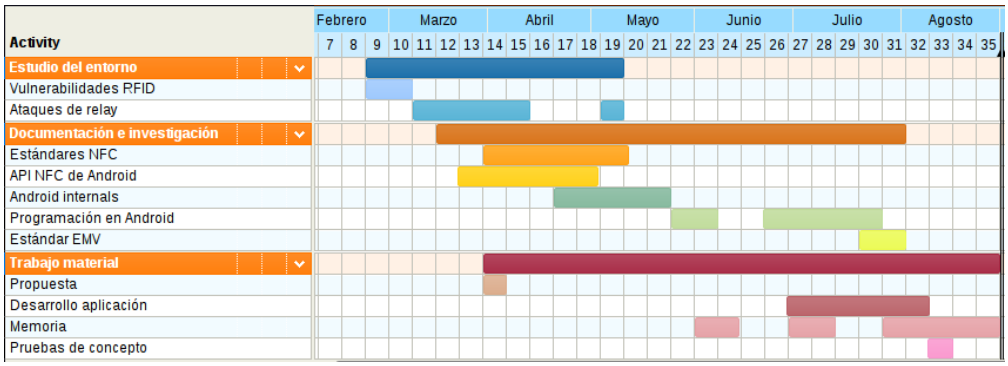


Figura A.1: Diagrama de Gantt mostrando el esfuerzo invertido en semanas.

Apéndice B

Mensajes de configuración NCI

CORE_SET_CONFIG_CMD				
Payload Field(s)	**Length**	**Value/description**		
Number of parameteres	1 octet	The number of parameter fields to follow (n)		
Parameter[1..n]	m+2 octets	ID	1 octet	The identifier of configuration parameter. See B.3 for a list of IDs.
		Len	1 octet	The length of Val(m). If Len is equal to 0x00, then the Val field is omitted, and the NFCC SHALL set the configuration parameter to its default value.
		Val	m octets	The value of the configuration paramter.

CORE_SET_CONFIG_RSP		
Payload Field(s)	**Length**	**Value/description**
Status	1 octet	See B.4.
Number of parameters	1 octet	The number of parameter ID fields to follow (n). Value SHALL be 0x00 and no parameter IDs listed unless Status == STATUS_INVAILD_PARAM.
Paramter ID [0..n]	1 octet	The identifier of the invalid configuration paramter. See B.3 for a list of IDs.

Tabla B.1: Mensajes de control usados por el DH para definir los parámetros del NFCC. Extraído del NFC Forum.

CORE_GET_CONFIG_CMD		
Payload Field(s)	**Length**	**Value/description**
Number of parameters	1 octet	The number of parameter ID fields to follow (n). Value SHALL be 0x00 and no parameter IDs listed unless Status == STATUS_INVAILD_PARAM.
Paramter ID [0..n]	1 octet	The identifier of configuration paramter. See B.3 for a list of IDs.

CORE_GET_CONFIG_RSP				
Payload Field(s)	**Length**	**Value/description**		
Status	1 octet	See B.4.		
Number of parameteres	1 octet	The number of parameter fields to follow (n)		
Parameter[1..n]	m+2 octets	ID	1 octet	The identifier of configuration parameter. See B.3 for a list of IDs.
		Len	1 octet	The length of Val(m). If Len is equal to 0x00, then the Val field is omitted.
		Val	m octets	The value of the configuration paramter.

Tabla B.2: Mensajes de control usados por el DH para leer los parámetros del NFCC. Extraído de la "NCI Technical Specification"del NFC Forum.

Tabla B.3: Parámetros de configuración. Extraído de la "NCI Technical Specification"del NFC Forum.

Parameter Name	**Tag**
Common Discovery Parameters	
TOTAL_DURATION	0x00
CON_DEVICES_LIMIT	0x01
CON_DISCOVERY_PARAM	0x02
RFU	0x03-0x07
Poll Mode – NFC-A Discovery Parameters	
PA_BAIL_OUT	0x08
RFU	0x09-0x0F
Poll Mode – NFC-B Discovery Parameters	
PB_AFI	0x10
PB_BAIL_OUT	0x11
PB_ATTRIB_PARAM1	0x12
PB_SENSB_REQ_PARAM	0x13
RFU	0x14-0x17
Poll Mode – NFC-F Discovery Parameters	
PF_BIT_RATE	0x18
PF_RC_CODE	0x19
RFU	0x1A-0x1F
Poll Mode – ISO-DEP Discovery Parameters	
PB_H_INFO	0x20
PI_BIT_RATE	0x21
RFU	0x22-0x27
Poll Mode – NFC-DEP Discovery Parameters	
PN_NFC_DEP_SPEED	0x28
PN_ATR_REQ_GEN_BYTES	0x29
PN_ATR_REQ_CONFIG	0x2A
RFU	0x2B-0x2F
Listen Mode – NFC-A Discovery Parameters	
LA_BIT_FRAME_SDD	0x30
LA_PLATFORM_CONFIG	0x31
LA_SEL_INFO	0x32
LA_NFCID1	0x33
RFU	0x34-0x37
Listen Mode – NFC-B Discovery Parameters	
LB_SENSB_INFO	0x38
LB_NFCID0	0x39
LB_APPLICATION_DATA	0x3A
LB_SFGI	0x3B
LB_ADC_FO	0x3C

RFU	0x3D - 0x3F
Listen Mode – NFC-F Discovery Parameters	
LF_T3T_IDENTIFIERS_1	0x40
LF_T3T_IDENTIFIERS_2	0x41
LF_T3T_IDENTIFIERS_3	0x42
LF_T3T_IDENTIFIERS_4	0x43
LF_T3T_IDENTIFIERS_5	0x44
LF_T3T_IDENTIFIERS_6	0x45
LF_T3T_IDENTIFIERS_7	0x46
LF_T3T_IDENTIFIERS_8	0x47
LF_T3T_IDENTIFIERS_9	0x48
LF_T3T_IDENTIFIERS_10	0x49
LF_T3T_IDENTIFIERS_11	0x4A
LF_T3T_IDENTIFIERS_12	0x4B
LF_T3T_IDENTIFIERS_13	0x4C
LF_T3T_IDENTIFIERS_14	0x4D
LF_T3T_IDENTIFIERS_15	0x4E
LF_T3T_IDENTIFIERS_16	0x4F
LF_PROTOCOL_TYPE	0x50
LF_T3T_PMM_DEFAULT	0x51
LF_T3T_MAX	0x52
LF_T3T_FLAGS	0x53
LF_CON_BITR_F	0x54
LF_ADV_FEAT	0x55
RFU	0x56-0x57
Listen Mode – ISO-DEP Discovery Parameters	
LI_FWI	0x58
LA_HIST_BY	0x59
LB_H_INFO_RESP	0x5A
LI_BIT_RATE	0x5B
RFU	0x5C-0x5F
Listen Mode – NFC-DEP Discovery Parameters	
LN_WT	0x60
LN_ATR_RES_GEN_BYTES	0x61
LN_ATR_RES_CONFIG	0x62
RFU	0x63-0x7F
Other Parameters	
RF_FIELD_INFO	0x80
RF_NFCEE_ACTION	0x81
NFCDEP_OP	0x82
LLCP_VERSION	0x83
AGG_INTF_CONFIG	0x84
NFCC_CONFIG_CONTROL	0x85

RFU	0x86-0x9F
Reserved for Proprietary Use	
Reserved	0xA0-0xFE
Reserved for Extension	
RFU	0xFF

Tabla B.4: Códigos de estado. Extraído de la "NCI Technical Specification"del NFC Forum.

Status code	Description
	Generic Status Codes
0x00	STATUS_OK
0x01	STATUS_REJECTED
0x03	STATUS_FAILED
0x04	STATUS_NOT_INITIALIZED
0x05	STATUS_SYNTAX_ERROR
0x06	STATUS_SEMANTIC_ERROR
0x07-0x08	RFU
0x09	STATUS_INVALID_PARAM
0x0A	STATUS_MESSAGE_SIZE_EXCEEDED
0x0B-0x10	RFU
0x11	STATUS_OK_1_BIT
0x12	STATUS_OK_2_BIT
0x13	STATUS_OK_3_BIT
0x14	STATUS_OK_4_BIT
0x15	STATUS_OK_5_BIT
0x16	STATUS_OK_6_BIT
0x17	STATUS_OK_7_BIT
0x18-0x9F	RFU
	RF Discovery Specific Status Codes
0xA0	DISCOVERY_ALREADY_STARTED
0xA1	DISCOVERY_TARGET_ACTIVATION_FAILED
0xA2	DISCOVERY_TEAR_DOWN
0xA3-	0xAF RFU
	RF Interface Specific Status Codes
0x02	RF_FRAME_CORRUPTED
0xB0	RF_TRANSMISSION_ERROR
0xB1	RF_PROTOCOL_ERROR
0xB2	RF_TIMEOUT_ERROR
0xB3	RF_UNEXPECTED_DATA
0xB4-0xBF	RFU
	NFCEE Interface Specific Status Codes
0xC0	NFCEE_INTERFACE_ACTIVATION_FAILED
0xC1	NFCEE_TRANSMISSION_ERROR
0xC2	NFCEE_PROTOCOL_ERROR
0xC3	NFCEE_TIMEOUT_ERROR
0xC4-0xDF	RFU
	Proprietary Status Codes
0xE0-0xFF	For proprietary use

Apéndice C

Traza de una llamada `transceive` en Android con NfcA

En este apartado se detalla la implementación a través de las distintas llamadas que se ejecutan en Android a partir del método `transceive` de un tag con tecnología NfcA. De este modo el lector podrá sumergirse en el código de bajo nivel con mayor facilidad, ya que conseguirá una visión global del proceso.

La clase `Tag` es la base que expone la API para comunicarse con una tarjeta cercana. Tiene un campo `INfcTag mTagService` de tipo `IBinder` (una interfaz a un objeto remoto en Android), que es usado por la clase `BasicTechnology` sobre la que se implementan todas las tecnologías del siguiente modo:

Código C.1: Tag.java

```
byte[] transceive(byte[] data, boolean raw) {
...
mTag.getTagService().transceive(mTag.getServiceHandle(), data, raw);
...
}
```

Nota: El booleano `raw` es ignorado en la implementación de `libnfc-nci`, mientras que en `libnfc-nxp` la utilizan para el envío de órdenes a las tarjetas Mifare Classic.

De modo que al llamar a `transceive` en realidad se realiza una invocación remota mediante IPC (la propia clase `INfcTag` es autogenerada a partir de un fichero AIDL[1], cuya implementación se encuentra en la clase `TagService` de `NfcService.java`[2] que es el servicio NFC del sistema. La Figura C.1 describe la implementación descrita.

`TagService` mantiene un objecto `TagEndpoint` que referencia a la tarjeta remota, dicha clase es tan sólo una interfaz en `DeviceHost.java`[3] y su implementación va a

[1] https://android.googlesource.com/platform/frameworks/base/+/android-4.4.1_r1.0.1/core/java/android/nfc/INfcTag.aidl

[2] https://android.googlesource.com/platform/packages/apps/Nfc/+/android-4.4.1_r1.0.1/src/com/android/nfc/NfcService.java

[3] https://android.googlesource.com/platform/packages/apps/Nfc/+/android-4.4.1_r1.0.1/src/com/android/nfc/DeviceHost.java

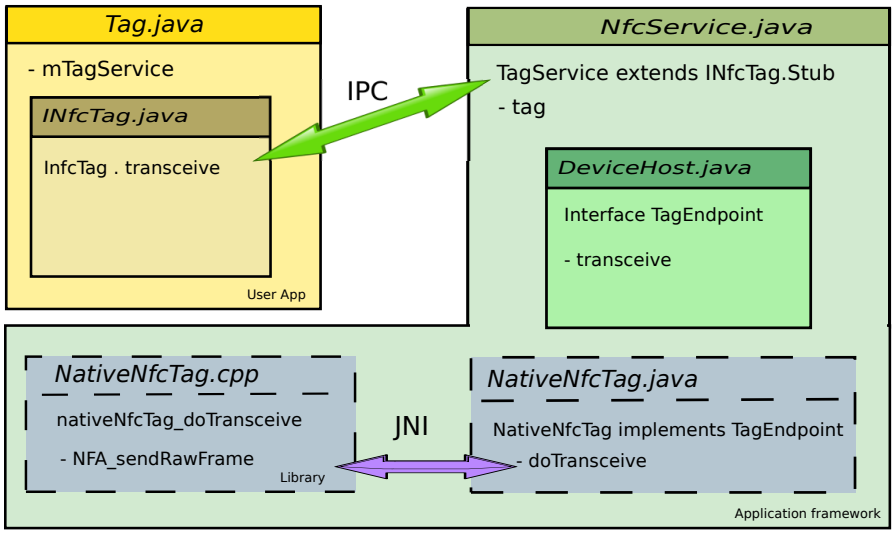


Figura C.1: Arquitectura de la implementación `Tag` en la API de Android.

depender del fabricante, según use `libnfc-nxp` o `libnfc-nci`. En nuestro caso, encontramos la implementación en `NativeNfcTag.java` que va a utilizar JNI para llamar al método `doNative` implementando en `NativeNfcTag.cpp`.

A partir de aquí se entra en otro nivel, donde se puede ver el predicado de la función en C++:

Código C.2: NativeNfcTag.cpp

```
static jbyteArray nativeNfcTag_doTransceive (JNIEnv* e, jobject, jbyteArray data,
    jboolean raw, jintArray statusTargetLost);
```

Tras comprobar el estado de la tarjeta y definir un *timeout* se invoca a la función `NFA_SendRawFrame`, pasando el *buffer* con el *payload* original. Este proceso se repite a lo largo de varios niveles descendiendo por los distintos módulos.

Al tener varios hilos y capas, la librería `libnfc-nci` implementa un mecanismo de colas de eventos y paso de mensajes fiable llamado GKI (General Kernal Interface) [4], que sirve de abstracción entre capas permitiendo una mayor independencia entre módulos. Básicamente cada tarea tiene un *buffer* (o buzón de entrada) de mensajes, estos mensajes serán procesados según su campo de evento y así el *payload* irá avanzando hasta el NFCC. Esta arquitectura facilita la gestión asíncrona del proceso.

Código C.3: Traza de una operación transceive.

```
D/USERIAL_LINUX(  870): my_read: return 27(0x1b) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f865ec, count=1, length=27
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f86bd0, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
I/NfcNciHal(  870): nfc_hal_dm_set_nfc_wake () ASSERT
D/USERIAL_LINUX(  870): UPIO_Set: ioctl, state=0
D/USERIAL_LINUX(  870): UPIO_Set: ioctl, old state=1, insert delay for 20 ms
```

[4] https://android.googlesource.com/platform/external/libnfc-nci/+/master/src/gki/

```
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=26
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=26
I/BrcmNfcNfa(  870): NFC received ntf gid:1
I/BrcmNfcNfa(  870): nfc_set_state 4 (IDLE)->5 (OPEN)
I/BrcmNfcNfa(  870): nfc_ncif_proc_activate:23 0, mode:0x00
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4004
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: DISCOVERY (1), event: ACTIVATED_NTF(5)
    disc_flags: 0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_new_state (): old_state: DISCOVERY (1), new_state: POLL_ACTIVE
     (4) disc_flags: 0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_notify_activation (): tech_n_mode:0x0, proto:0x2
I/BrcmNfcNfa(  870): nfa_dm_disc_get_disc_mask (): tech_n_mode:0x0, protocol:0x2, disc_mask:0x2
I/BrcmNfcNfa(  870): activated_protocol:0x2, activated_handle: 0x1
I/BrcmNfcNfa(  870): nfa_dm_poll_disc_cback (): event:0x01
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_ACTIVATE_NTF_EVT (0x601), flags:
    00000001
I/BrcmNfcNfa(  870): nfa_rw_activate_ntf
I/BrcmNfcNfa(  870): RW_SetActivatedTagType protocol:2, technology:0, SAK:0
I/BrcmNfcNfa(  870): nfa_dm_notify_activation_status (): status:0x0
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 5
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_ACTIVATED_EVT: gIsSelectingRfInterface=0,
    sIsDisabling=0
D/BrcmNfcJni(  870): NfcTag::setActivationState: state=2
D/BrcmNfcJni(  870): pn544InteropIsBusy: 0
D/BrcmNfcJni(  870): NfcTag::IsSameKovio: enter
D/BrcmNfcJni(  870): NfcTag::discoverTechnologies (activation): enter
D/BrcmNfcJni(  870): NfcTag::discoverTechnologies (activation): index=0; tech=1; handle=1; nfc
    type=2
D/BrcmNfcJni(  870): NfcTag::discoverTechnologies (activation): index=1; tech=9; handle=1; nfc
    type=2
D/BrcmNfcJni(  870): NfcTag::discoverTechnologies (activation): exit
D/BrcmNfcJni(  870): NfcTag::createNativeNfcTag: enter
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers1
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers2
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers3: index=0; rf tech params mode=0
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers3: tech A
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers3: index=1; rf tech params mode=0
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers3: tech A
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers4: index=0
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers4: T2T; tech A
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers4: index=1
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers4: T2T; tech A
D/BrcmNfcJni(  870): NfcTag::fillNativeNfcTagMembers5: tech A
D/BrcmNfcJni(  870): NfcTag::createNativeNfcTag: try notify nfc service
D/audio_hw_primary(  175): out_set_parameters: enter: usecase(1: low-latency-playback) kvpairs:
     routing=2
D/NativeNfcTag(  870): Connect to a tech with a different handle
D/BrcmNfcJni(  870): nativeNfcTag_doHandleReconnect: targetHandle = 0
D/BrcmNfcJni(  870): nativeNfcTag_doConnect: targetHandle = 0
D/BrcmNfcJni(  870): nativeNfcTag_doConnect() Nfc type = 2, do nothing for non ISO_DEP
D/BrcmNfcJni(  870): nativeNfcTag_doConnect: exit 0x0
D/BrcmNfcJni(  870): nativeNfcTag_doIsIsoDepNdefFormatable
D/BrcmNfcJni(  870): NfcTag::isMifareUltralight: return=1
D/BrcmNfcJni(  870): nativeNfcTag_doIsNdefFormatable: is formattable=1
D/BrcmNfcJni(  870): nativeNfcTag_doReconnect: enter
D/BrcmNfcJni(  870): reSelect: enter; rf intf = 1, current intf = 1
D/BrcmNfcJni(  870): reSelect: deactivate to sleep
I/BrcmNfcNfa(  870): NFA_Deactivate (): sleep_mode:1
D/BrcmNfcJni(  870): NfcTag::createNativeNfcTag: exit
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: POLL_ACTIVE (4), disc_flags: 0x1
I/BrcmNfcNfa(  870): NFA got event 0x0113
I/BrcmNfcNfa(  870): nfa_dm_evt_hdlr event: NFA_DM_API_DEACTIVATE_EVT (0x13)
I/BrcmNfcNfa(  870): nfa_dm_act_deactivate ()
D/audio_hw_primary(  175): select_devices: out_snd_device(2: speaker) in_snd_device(0: )
D/ACDB-LOADER(  175): ACDB -> send_afe_cal
I/BrcmNfcNfa(  870): nfa_dm_rf_deactivate () deactivate_type:0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: POLL_ACTIVE (4), event: DEACTIVATE_CMD
    (6) disc_flags: 0x1
I/BrcmNfcNfa(  870): NFC_Deactivate 5 (OPEN) deactivate_type:1
I/BrcmNfcNfa(  870): nfc_set_state 5 (OPEN)->6 (CLOSING)
I/BrcmNfcNfa(  870): act_protocol 2 credits:1/1
D/NfcAdaptation(  870): NfcAdaptation::HalWrite
D/NfcNciHal(  870): HaiWrite: enter; len=4
I/NfcNciHal(  870): HAL_NfcWrite ()
D/NfcNciHal(  870): HaiWrite: exit 0
I/NfcNciHal(  870): nfc_hal_main_send_message() ls:0x10
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 0
I/NfcNciHal(  870): p_buf->len: 4
D/USERIAL_LINUX(  870): USERIAL_Write: (5 bytes)
D/USERIAL_LINUX(  870): USERIAL_Write len = 5, ret = 5
I/BrcmNfcNfa(  870): nfa_sys_ptim_start_timer 74df0b5c
I/BrcmNfcNfa(  870): ptim timer start
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: POLL_ACTIVE (4), disc_flags: 0x61
```

```
D/USERIAL_LINUX(  870): my_read: return 5(0x5) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f86bc8, count=1, length=5
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f871ac, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/USERIAL_LINUX(  870): my_read: return 6(0x6) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f871a4, count=1, length=6
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f86144, len = 280
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=4
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=4
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=5
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=5
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
I/BrcmNfcNfa(  870): NFC received rsp gid:1
I/BrcmNfcNfa(  870): nfc_set_state 6 (CLOSING)->4 (IDLE)
I/BrcmNfcNfa(  870): rw_t2t_conn_cback: conn_id=0, evt=24578
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4005
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: POLL_ACTIVE (4), event: DEACTIVATE_RSP
    (7) disc_flags: 0x61
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: POLL_ACTIVE (4), disc_flags: 0x41
I/BrcmNfcNfa(  870): NFC received ntf gid:1
I/BrcmNfcNfa(  870): nfc_set_state 4 (IDLE)->4 (IDLE)
I/BrcmNfcNfa(  870): nfa_dm_disc_data_cback ()
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4005
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: POLL_ACTIVE (4), event: DEACTIVATE_NTF
    (8) disc_flags: 0x41
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df0b5c
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): nfa_dm_disc_notify_deactivation (): activated_handle=1
I/BrcmNfcNfa(  870): nfa_dm_poll_disc_cback (): event:0x02
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_DEACTIVATE_NTF_EVT (0x602), flags:
    00000021
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 6
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_DEACTIVATED_EVT   Type: 1, gIsTagDeactivating:
    1
D/BrcmNfcJni(  870): NfcTag::setDeactivationState: state=1
I/BrcmNfcNfa(  870): nfa_dm_disc_new_state (): old_state: POLL_ACTIVE (4), new_state:
    W4_HOST_SELECT (3) disc_flags: 0x1
D/BrcmNfcJni(  870): reSelect: select interface 1
I/BrcmNfcNfa(  870): NFA_Select (): rf_disc_id:0x1, protocol:0x2, rf_interface:0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: W4_HOST_SELECT (3), disc_flags: 0x1
I/BrcmNfcNfa(  870): NFA got event 0x0111
I/BrcmNfcNfa(  870): nfa_dm_evt_hdlr event: NFA_DM_API_SELECT_EVT (0x11)
I/BrcmNfcNfa(  870): nfa_dm_act_select ()
I/BrcmNfcNfa(  870): nfa_dm_disc_select () rf_disc_id:0x1, protocol:0x2, rf_interface:0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: W4_HOST_SELECT (3), event: SELECT_CMD(3)
     disc_flags: 0x11
D/NfcAdaptation(  870): NfcAdaptation::HalWrite
D/NfcNciHal(  870): HaiWrite: enter; len=6
I/NfcNciHal(  870): HAL_NfcWrite ()
I/NfcNciHal(  870): nfc_hal_main_send_message() ls:0x10
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 0
I/NfcNciHal(  870): p_buf->len: 6
D/USERIAL_LINUX(  870): USERIAL_Write: (7 bytes)
D/NfcNciHal(  870): HaiWrite: exit 0
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: W4_HOST_SELECT (3), disc_flags: 0x11
D/USERIAL_LINUX(  870): USERIAL_Write len = 7, ret = 7
I/BrcmNfcNfa(  870): ptim timer stop
D/USERIAL_LINUX(  870): my_read: return 5(0x5) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f8613c, count=1, length=5
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f86720, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=4
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=4
I/BrcmNfcNfa(  870): NFC received rsp gid:1
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4003
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: W4_HOST_SELECT (3), event: SELECT_RSP(4)
     disc_flags: 0x11
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 3
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_SELECT_RESULT_EVT: status = 0,
    gIsSelectingRfInterface = 1, sIsDisabling=0
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: W4_HOST_SELECT (3), disc_flags: 0x1
D/USERIAL_LINUX(  870): my_read: return 27(0x1b) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f86718, count=1, length=27
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f86978, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
```

```
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=26
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=26
I/BrcmNfcNfa(  870): NFC received ntf gid:1
I/BrcmNfcNfa(  870): nfc_set_state 4 (IDLE)->5 (OPEN)
I/BrcmNfcNfa(  870): nfc_ncif_proc_activate:23 0, mode:0x00
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4004
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: W4_HOST_SELECT (3), event: ACTIVATED_NTF
    (5) disc_flags: 0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_new_state (): old_state: W4_HOST_SELECT (3), new_state:
    POLL_ACTIVE (4) disc_flags: 0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_notify_activation (): tech_n_mode:0x0, proto:0x2
I/BrcmNfcNfa(  870): nfa_dm_disc_get_disc_mask (): tech_n_mode:0x0, protocol:0x2, disc_mask:0x2
I/BrcmNfcNfa(  870): activated_protocol:0x2, activated_handle: 0x1
I/BrcmNfcNfa(  870): nfa_dm_poll_disc_cback (): event:0x01
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_ACTIVATE_NTF_EVT (0x601), flags:
    00000001
I/BrcmNfcNfa(  870): nfa_rw_activate_ntf
I/BrcmNfcNfa(  870): RW_SetActivatedTagType protocol:2, technology:0, SAK:0
I/BrcmNfcNfa(  870): nfa_dm_notify_activation_status (): status:0x0
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 5
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_ACTIVATED_EVT: gIsSelectingRfInterface=1,
    sIsDisabling=0
D/BrcmNfcJni(  870): NfcTag::setActivationState: state=2
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: POLL_ACTIVE (4), disc_flags: 0x1
D/BrcmNfcJni(  870): reSelect: select completed; sConnectOk=1
D/BrcmNfcJni(  870): reSelect: exit; status=0
D/BrcmNfcJni(  870): nativeNfcTag_doReconnect: exit 0x0
D/BrcmNfcJni(  870): nativeNfcTag_doCheckNdef: enter
D/BrcmNfcJni(  870): nativeNfcTag_doCheckNdef: try NFA_RwDetectNDef
I/BrcmNfcNfa(  870): NFA_RwDetectNDef
I/BrcmNfcNfa(  870): NFA got event 0x0600
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_OP_REQUEST_EVT (0x600), flags: 00000021
I/BrcmNfcNfa(  870): nfa_rw_handle_op_req: op=0x00
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
I/BrcmNfcNfa(  870): nfa_rw_detect_ndef
I/BrcmNfcNfa(  870): RW SENT []:0x30 CMD
I/BrcmNfcNfa(  870): nfc_ncif_send_data :0, num_buff:1 qc:0
D/NfcAdaptation(  870): NfcAdaptation::HalWrite
D/NfcNciHal(  870): HaiWrite: enter; len=5
I/NfcNciHal(  870): HAL_NfcWrite ()
D/NfcNciHal(  870): HaiWrite: exit 0
I/NfcNciHal(  870): nfc_hal_main_send_message() ls:0x0
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 0
D/USERIAL_LINUX(  870): USERIAL_Write: (6 bytes)
I/BrcmNfcNfa(  870): rw_t2t_read Sent Command for Block: 0
D/USERIAL_LINUX(  870): USERIAL_Write len = 6, ret = 6
D/USERIAL_LINUX(  870): my_read: return 7(0x7) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f86970, count=1, length=7
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f864c8, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=6
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=6
I/BrcmNfcNfa(  870): NFC received ntf gid:0
I/BrcmNfcNfa(  870): nci_proc_core_ntf opcode:0x6
I/BrcmNfcNfa(  870): nfc_ncif_send_data :0, num_buff:1 qc:0
D/USERIAL_LINUX(  870): my_read: return 21(0x15) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f864c0, count=1, length=21
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f865f4, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=20
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=20
I/BrcmNfcNfa(  870): NFC received data
I/BrcmNfcNfa(  870): nfc_ncif_proc_data 0x000011
I/BrcmNfcNfa(  870): nfc_ncif_proc_data len:17
I/BrcmNfcNfa(  870): rw_t2t_conn_cback: conn_id=0, evt=24579
I/BrcmNfcNfa(  870): RW RECV []:0x30 RSP
I/BrcmNfcNfa(  870): rw_t2t_proc_data State: 5  conn_id: 0  event: 24579  len: 16  data[0]: 0
    x04
I/BrcmNfcNfa(  870): NDEF Detection failed!, CC[0]: 0x00, CC[1]: 0x00, CC[3]: 0x00
I/BrcmNfcNfa(  870): nfa_rw_cback: event=0x43
I/BrcmNfcNfa(  870): NDEF Detection completed: cur_size=0, max_size=46, flags=0x34
I/BrcmNfcNfa(  870): nfa_dm_act_conn_cback_notify (): event:0x8
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 8
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_NDEF_DETECT_EVT: status = 0x3, protocol = 2,
    max_size = 46, cur_size = 0, flags = 0x34
D/BrcmNfcJni(  870): nativeNfcTag_doCheckNdefResult: flag ndef supported
D/BrcmNfcJni(  870): nativeNfcTag_doCheckNdefResult: flag formattable
I/BrcmNfcNfa(  870): RW T2T state changed:<TLV_DETECT> -> <IDLE>
```

```
D/BrcmNfcJni(  870): nativeNfcTag_doCheckNdef: exit; status=0x3
D/NativeNfcTag(  870): Check NDEF Failed - status = 3
D/BrcmNfcJni(  870): nativeNfcTag_doHandleReconnect: targetHandle = 1
D/BrcmNfcJni(  870): nativeNfcTag_doConnect: targetHandle = 1
D/BrcmNfcJni(  870): nativeNfcTag_doConnect() Nfc type = 2, do nothing for non ISO_DEP
D/BrcmNfcJni(  870): nativeNfcTag_doConnect: exit 0x0
D/BrcmNfcJni(  870): nativeNfcTag_doCheckNdef: enter
D/BrcmNfcJni(  870): nativeNfcTag_doCheckNdef: try NFA_RwDetectNDef
I/BrcmNfcNfa(  870): NFA_RwDetectNDef
I/BrcmNfcNfa(  870): NFA got event 0x0600
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_OP_REQUEST_EVT (0x600), flags: 00000021
I/BrcmNfcNfa(  870): nfa_rw_handle_op_req: op=0x00
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
I/BrcmNfcNfa(  870): nfa_rw_detect_ndef
I/BrcmNfcNfa(  870): RW_T2tLocateTlv - Invalid NDEF Magic Number!, CC[0]: 0x00, CC[1]: 0x00, CC
    [3]: 0x00
I/BrcmNfcNfa(  870): nfa_dm_act_conn_cback_notify (): event:0x8
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 8
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_NDEF_DETECT_EVT: status = 0x3, protocol = 0,
    max_size = 0, cur_size = 0, flags = 0x8
D/BrcmNfcJni(  870): nativeNfcTag_doCheckNdefResult: flag all unknown
D/BrcmNfcJni(  870): nativeNfcTag_doCheckNdef: exit; status=0x3
D/NativeNfcTag(  870): Check NDEF Failed - status = 3
D/BrcmNfcJni(  870): nativeNfcTag_doReconnect: enter
D/BrcmNfcJni(  870): reSelect: enter; rf intf = 1, current intf = 1
D/BrcmNfcJni(  870): reSelect: deactivate to sleep
I/BrcmNfcNfa(  870): NFA_Deactivate (): sleep_mode:1
I/BrcmNfcNfa(  870): NFA got event 0x0113
I/BrcmNfcNfa(  870): nfa_dm_evt_hdlr event: NFA_DM_API_DEACTIVATE_EVT (0x13)
I/BrcmNfcNfa(  870): nfa_dm_act_deactivate ()
I/BrcmNfcNfa(  870): nfa_dm_rf_deactivate () deactivate_type:0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: POLL_ACTIVE (4), event: DEACTIVATE_CMD
    (6) disc_flags: 0x1
I/BrcmNfcNfa(  870): NFC_Deactivate 5 (OPEN) deactivate_type:1
I/BrcmNfcNfa(  870): nfc_set_state 5 (OPEN)->6 (CLOSING)
I/BrcmNfcNfa(  870): act_protocol 2 credits:1/1
D/NfcAdaptation(  870): NfcAdaptation::HalWrite
D/NfcNciHal(  870): HaiWrite: enter; len=4
I/NfcNciHal(  870): HAL_NfcWrite ()
D/NfcNciHal(  870): HaiWrite: exit 0
I/BrcmNfcNfa(  870): nfa_sys_ptim_start_timer 74df0b5c
I/BrcmNfcNfa(  870): ptim timer start
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: POLL_ACTIVE (4), disc_flags: 0x61
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
I/NfcNciHal(  870): nfc_hal_main_send_message() ls:0x10
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 0
I/NfcNciHal(  870): p_buf->len: 4
D/USERIAL_LINUX(  870): USERIAL_Write: (5 bytes)
D/USERIAL_LINUX(  870): doWriteDelay() delay 1 ms
D/USERIAL_LINUX(  870): USERIAL_Write len = 5, ret = 5
D/USERIAL_LINUX(  870): my_read: return 5(0x5) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f865ec, count=1, length=5
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f86bd0, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=4
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=4
I/BrcmNfcNfa(  870): NFC received rsp gid:1
I/BrcmNfcNfa(  870): nfc_set_state 6 (CLOSING)->4 (IDLE)
I/BrcmNfcNfa(  870): rw_t2t_conn_cback: conn_id=0, evt=24578
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4005
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: POLL_ACTIVE (4), event: DEACTIVATE_RSP
    (7) disc_flags: 0x61
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: POLL_ACTIVE (4), disc_flags: 0x41
D/USERIAL_LINUX(  870): my_read: return 6(0x6) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f86bc8, count=1, length=6
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f871ac, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=5
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=5
I/BrcmNfcNfa(  870): NFC received ntf gid:1
I/BrcmNfcNfa(  870): nfc_set_state 4 (IDLE)->4 (IDLE)
I/BrcmNfcNfa(  870): nfa_dm_disc_data_cback ()
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4005
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: POLL_ACTIVE (4), event: DEACTIVATE_NTF
    (8) disc_flags: 0x41
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df0b5c
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): nfa_dm_disc_notify_deactivation (): activated_handle=1
```

```
I/BrcmNfcNfa(  870): nfa_dm_poll_disc_cback (): event:0x02
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_DEACTIVATE_NTF_EVT (0x602), flags:
    00000021
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 6
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_DEACTIVATED_EVT   Type: 1, gIsTagDeactivating:
    1
D/BrcmNfcJni(  870): NfcTag::setDeactivationState: state=1
D/BrcmNfcJni(  870): reSelect: select interface 1
I/BrcmNfcNfa(  870): NFA_Select (): rf_disc_id:0x1, protocol:0x2, rf_interface:0x1
I/BrcmNfcNfa(  870): NFA_Select (): rf_disc_id:0x1, protocol:0x2, rf_interface:0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: W4_HOST_SELECT (3), disc_flags: 0x1
I/BrcmNfcNfa(  870): NFA got event 0x0111
I/BrcmNfcNfa(  870): nfa_dm_evt_hdlr event: NFA_DM_API_SELECT_EVT (0x11)
I/BrcmNfcNfa(  870): nfa_dm_act_select ()
I/BrcmNfcNfa(  870): nfa_dm_disc_select () rf_disc_id:0x1, protocol:0x2, rf_interface:0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: W4_HOST_SELECT (3), event: SELECT_CMD(3)
     disc_flags: 0x11
D/NfcAdaptation(  870): NfcAdaptation::HalWrite
D/NfcNciHal(  870): HaiWrite: enter; len=6
I/NfcNciHal(  870): HAL_NfcWrite ()
D/NfcNciHal(  870): HaiWrite: exit 0
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: W4_HOST_SELECT (3), disc_flags: 0x11
I/NfcNciHal(  870): nfc_hal_main_send_message() ls:0x10
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 0
I/NfcNciHal(  870): p_buf->len: 6
D/USERIAL_LINUX(  870): USERIAL_Write: (7 bytes)
D/USERIAL_LINUX(  870): doWriteDelay() delay 2 ms
D/USERIAL_LINUX(  870): USERIAL_Write len = 7, ret = 7
D/USERIAL_LINUX(  870): my_read: return 5(0x5) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f871a4, count=1, length=5
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f86144, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=4
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=4
I/BrcmNfcNfa(  870): NFC received rsp gid:1
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4003
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: W4_HOST_SELECT (3), event: SELECT_RSP(4)
     disc_flags: 0x11
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 3
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_SELECT_RESULT_EVT: status = 0,
    gIsSelectingRfInterface = 1, sIsDisabling=0
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: W4_HOST_SELECT (3), disc_flags: 0x1
D/USERIAL_LINUX(  870): my_read: return 27(0x1b) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f8613c, count=1, length=27
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f86720, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=26
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=26
I/BrcmNfcNfa(  870): NFC received ntf gid:1
I/BrcmNfcNfa(  870): nfc_set_state 4 (IDLE)->5 (OPEN)
I/BrcmNfcNfa(  870): nfc_ncif_proc_activate:23 0, mode:0x00
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4004
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: W4_HOST_SELECT (3), event: ACTIVATED_NTF
    (5) disc_flags: 0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_new_state (): old_state: W4_HOST_SELECT (3), new_state:
    POLL_ACTIVE (4) disc_flags: 0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_notify_activation (): tech_n_mode:0x0, proto:0x2
I/BrcmNfcNfa(  870): nfa_dm_disc_get_disc_mask (): tech_n_mode:0x0, protocol:0x2, disc_mask:0x2
I/BrcmNfcNfa(  870): activated_protocol:0x2, activated_handle: 0x1
I/BrcmNfcNfa(  870): nfa_dm_poll_disc_cback (): event:0x01
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_ACTIVATE_NTF_EVT (0x601), flags:
    00000001
I/BrcmNfcNfa(  870): nfa_rw_activate_ntf
I/BrcmNfcNfa(  870): RW_SetActivatedTagType protocol:2, technology:0, SAK:0
I/BrcmNfcNfa(  870): nfa_dm_notify_activation_status (): status:0x0
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 5
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_ACTIVATED_EVT: gIsSelectingRfInterface=1,
    sIsDisabling=0
D/BrcmNfcJni(  870): NfcTag::setActivationState: state=2
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: POLL_ACTIVE (4), disc_flags: 0x1
D/BrcmNfcJni(  870): reSelect: select completed; sConnectOk=1
D/BrcmNfcJni(  870): reSelect: exit; status=0
D/BrcmNfcJni(  870): nativeNfcTag_doReconnect: exit 0x0
D/BrcmNfcJni(  870): nativeNfcTag_doTransceive: enter; raw=0; timeout = 618
I/BrcmNfcNfa(  870): NFA_SendRawFrame () data_len:2
I/BrcmNfcNfa(  870): NFA got event 0x010C
I/BrcmNfcNfa(  870): nfa_dm_evt_hdlr event: NFA_DM_API_RAW_FRAME_EVT (0x0c)
I/BrcmNfcNfa(  870): nfa_dm_act_send_raw_frame ()
```

```
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_OP_REQUEST_EVT (0x600), flags: 00000021
I/BrcmNfcNfa(  870): nfa_rw_handle_op_req: op=0x05
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
I/BrcmNfcNfa(  870): nfc_ncif_send_data :0, num_buff:1 qc:0
D/NfcAdaptation(  870): NfcAdaptation::HalWrite
D/NfcNciHal(  870): HaiWrite: enter; len=5
I/NfcNciHal(  870): HAL_NfcWrite ()
D/NfcNciHal(  870): HaiWrite: exit 0
I/NfcNciHal(  870): nfc_hal_main_send_message() ls:0x0
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 0
D/USERIAL_LINUX(  870): USERIAL_Write: (6 bytes)
D/NativeNfcTag(  870): Starting background presence check
D/USERIAL_LINUX(  870): USERIAL_Write len = 6, ret = 6
D/USERIAL_LINUX(  870): my_read: return 7(0x7) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f86718, count=1, length=7
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f86978, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDatadCallback: enter; len=6
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=6
I/BrcmNfcNfa(  870): NFC received ntf gid:0
I/BrcmNfcNfa(  870): nci_proc_core_ntf opcode:0x6
I/BrcmNfcNfa(  870): nfc_ncif_send_data :0, num_buff:1 qc:0
D/USERIAL_LINUX(  870): my_read: return 21(0x15) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f86970, count=1, length=21
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=20
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=20
I/BrcmNfcNfa(  870): NFC received data
I/BrcmNfcNfa(  870): nfc_ncif_proc_data 0x000011
I/BrcmNfcNfa(  870): nfc_ncif_proc_data len:17
I/BrcmNfcNfa(  870): rw_t2t_conn_cback: conn_id=0, evt=24579
I/BrcmNfcNfa(  870): rw_t2t_proc_data - Raw frame event! state: IDLE, conn_id: 0  event: 24579
I/BrcmNfcNfa(  870): nfa_rw_cback: event=0x48
I/BrcmNfcNfa(  870): nfa_dm_act_conn_cback_notify (): event:0x9
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 9
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_DATA_EVT:  len = 16
D/BrcmNfcJni(  870): nativeNfcTag_doTransceiveStatus: data len=16, waiting for transceive: 1
D/BrcmNfcJni(  870): nativeNfcTag_doTransceive: response 16 bytes
D/BrcmNfcJni(  870): NfcTag::isT2tNackResponse: return 0
D/BrcmNfcJni(  870): nativeNfcTag_doTransceive: exit
D/NfcDispatcher(  870): dispatch tag: TAG: Tech [android.nfc.tech.NfcA, android.nfc.tech.
    MifareUltralight, android.nfc.tech.NdefFormatable] message: null
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f86270, len = 280
I/NfcDispatcher(  870): matched TECH override
D/NfcDispatcher(  870): Set Foreground Dispatch
I/Foreground dispatch( 7783): Discoverred tag with intent: Intent { act=android.nfc.action.
    TECH_DISCOVERED flg=0x20000000 cmp=com.example.nfctest/.MainActivity (has extras) }
D/BrcmNfcJni(  870): nativeNfcTag_doTransceive: enter; raw=1; timeout = 618
I/BrcmNfcNfa(  870): NFA_SendRawFrame () data_len:7
I/BrcmNfcNfa(  870): NFA got event 0x010C
I/BrcmNfcNfa(  870): nfa_dm_evt_hdlr event: NFA_DM_API_RAW_FRAME_EVT (0x0c)
I/BrcmNfcNfa(  870): nfa_dm_act_send_raw_frame ()
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_OP_REQUEST_EVT (0x600), flags: 00000021
I/BrcmNfcNfa(  870): nfa_rw_handle_op_req: op=0x05
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
I/BrcmNfcNfa(  870): nfc_ncif_send_data :0, num_buff:1 qc:0
D/NfcAdaptation(  870): NfcAdaptation::HalWrite
D/NfcNciHal(  870): HaiWrite: enter; len=10
I/NfcNciHal(  870): HAL_NfcWrite ()
D/NfcNciHal(  870): HaiWrite: exit 0
I/NfcNciHal(  870): nfc_hal_main_send_message() ls:0x0
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 0
D/USERIAL_LINUX(  870): USERIAL_Write: (11 bytes)
D/USERIAL_LINUX(  870): USERIAL_Write len = 11, ret = 11
D/USERIAL_LINUX(  870): my_read: return 7(0x7) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f86268, count=1, length=7
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f865f4, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=6
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=6
I/BrcmNfcNfa(  870): NFC received ntf gid:0
I/BrcmNfcNfa(  870): nci_proc_core_ntf opcode:0x6
I/BrcmNfcNfa(  870): nfc_ncif_send_data :0, num_buff:1 qc:0
I/NfcNciHal(  870): nci_brcm_lp_timeout_cback ()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 2
I/NfcNciHal(  870): nfc_hal_dm_set_nfc_wake () DEASSERT
```

```
D/USERIAL_LINUX(  870): UPIO_Set: ioctl, state=1
E/BrcmNfcJni(  870): nativeNfcTag_doTransceive: wait response timeout
D/BrcmNfcJni(  870): nativeNfcTag_doTransceive: exit
D/NfcDispatcher(  870): Set Foreground Dispatch
I/Choreographer( 7783): Skipped 37 frames!  The application may be doing too much work on its
    main thread.
D/BrcmNfcJni(  870): nativeNfcTag_doPresenceCheck
I/BrcmNfcNfa(  870): NFA_RwPresenceCheck
I/BrcmNfcNfa(  870): NFA got event 0x0600
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_OP_REQUEST_EVT (0x600), flags: 00000021
I/BrcmNfcNfa(  870): nfa_rw_handle_op_req: op=0x03
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
I/BrcmNfcNfa(  870): RW_T2tPresenceCheck
I/BrcmNfcNfa(  870): RW SENT []:0x30 CMD
I/BrcmNfcNfa(  870): nfc_ncif_send_data :0, num_buff:1 qc:0
D/NfcAdaptation(  870): NfcAdaptation::HalWrite
D/NfcNciHal(  870): HaiWrite: enter; len=5
I/NfcNciHal(  870): HAL_NfcWrite ()
D/NfcNciHal(  870): HaiWrite: exit 0
I/NfcNciHal(  870): nfc_hal_main_send_message() ls:0x0
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 0
I/NfcNciHal(  870): nfc_hal_dm_set_nfc_wake () ASSERT
D/USERIAL_LINUX(  870): UPIO_Set: ioctl, state=0
D/USERIAL_LINUX(  870): UPIO_Set: ioctl, old state=1, insert delay for 20 ms
D/USERIAL_LINUX(  870): USERIAL_Write: (6 bytes)
D/USERIAL_LINUX(  870): doWriteDelay() delay 19 ms
D/USERIAL_LINUX(  870): USERIAL_Write len = 6, ret = 6
D/USERIAL_LINUX(  870): my_read: return 7(0x7) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f865ec, count=1, length=7
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f86f54, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=6
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=6
I/BrcmNfcNfa(  870): NFC received ntf gid:0
I/BrcmNfcNfa(  870): nci_proc_core_ntf opcode:0x6
I/BrcmNfcNfa(  870): nfc_ncif_send_data :0, num_buff:1 qc:0
I/NfcNciHal(  870): nci_brcm_lp_timeout_cback ()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 2
I/NfcNciHal(  870): nfc_hal_dm_set_nfc_wake () DEASSERT
D/USERIAL_LINUX(  870): UPIO_Set: ioctl, state=1
I/BrcmNfcNfa(  870): nfa_rw_cback: event=0x49
I/BrcmNfcNfa(  870): nfa_dm_act_conn_cback_notify (): event:0xF
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 15
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_PRESENCE_CHECK_EVT
I/BrcmNfcNfa(  870): Presence check failed. Deactivating...
D/BrcmNfcJni(  870): nativeNfcTag_doPresenceCheck: tag absent
D/NativeNfcTag(  870): Tag lost, restarting polling loop
D/BrcmNfcJni(  870): nativeNfcTag_doDisconnect: enter
I/BrcmNfcNfa(  870): NFA_Deactivate (): sleep_mode:0
D/BrcmNfcJni(  870): nativeNfcTag_doDisconnect: exit
D/NativeNfcTag(  870): Stopping background presence check
I/BrcmNfcNfa(  870): nfa_dm_rf_deactivate () deactivate_type:0x3
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: POLL_ACTIVE (4), event: DEACTIVATE_CMD
    (6) disc_flags: 0x1
I/BrcmNfcNfa(  870): NFC_Deactivate 5 (OPEN) deactivate_type:3
I/BrcmNfcNfa(  870): nfc_set_state 5 (OPEN)->6 (CLOSING)
I/BrcmNfcNfa(  870): act_protocol 2 credits:1/1
D/NfcAdaptation(  870): NfcAdaptation::HalWrite
D/NfcNciHal(  870): HaiWrite: enter; len=4
I/NfcNciHal(  870): HAL_NfcWrite ()
D/NfcNciHal(  870): HaiWrite: exit 0
I/NfcNciHal(  870): nfc_hal_main_send_message() ls:0x10
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 0
I/NfcNciHal(  870): nfc_hal_dm_set_nfc_wake () ASSERT
D/USERIAL_LINUX(  870): UPIO_Set: ioctl, state=0
D/USERIAL_LINUX(  870): UPIO_Set: ioctl, old state=1, insert delay for 20 ms
I/NfcNciHal(  870): p_buf->len: 4
D/USERIAL_LINUX(  870): USERIAL_Write: (5 bytes)
D/USERIAL_LINUX(  870): doWriteDelay() delay 15 ms
I/BrcmNfcNfa(  870): nfa_sys_ptim_start_timer 74df0b5c
I/BrcmNfcNfa(  870): ptim timer start
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: POLL_ACTIVE (4), disc_flags: 0x61
I/BrcmNfcNfa(  870): NFA got event 0x0113
I/BrcmNfcNfa(  870): nfa_dm_evt_hdlr event: NFA_DM_API_DEACTIVATE_EVT (0x13)
I/BrcmNfcNfa(  870): nfa_dm_act_deactivate ()
I/BrcmNfcNfa(  870): nfa_dm_rf_deactivate () deactivate_type:0x3
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: POLL_ACTIVE (4), event: DEACTIVATE_CMD
    (6) disc_flags: 0x61
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: POLL_ACTIVE (4), disc_flags: 0x61
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
```

```
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
D/USERIAL_LINUX(  870): USERIAL_Write len = 5, ret = 5
D/USERIAL_LINUX(  870): my_read: return 5(0x5) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f86f4c, count=1, length=5
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f8684c, len = 280
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/USERIAL_LINUX(  870): my_read: return 6(0x6) bytes, errno=0 count=280, n=1, timeout=-1
D/USERIAL_LINUX(  870): userial_read_thread(): enqueued p_buf=0x74f86844, count=1, length=6
D/USERIAL_LINUX(  870): my_read: enter, pbuf=74f87080, len = 280
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=4
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=4
I/NfcNciHal(  870): nfc_hal_nci_preproc_rx_nci_msg()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 1
D/NfcNciHal(  870): BroadcomHalDataCallback: enter; len=5
D/NfcAdaptation(  870): NfcAdaptation::HalDeviceContextDataCallback: len=5
I/BrcmNfcNfa(  870): NFC received rsp gid:1
I/BrcmNfcNfa(  870): nfc_set_state 6 (CLOSING)->4 (IDLE)
I/BrcmNfcNfa(  870): rw_t2t_conn_cback: conn_id=0, evt=24578
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4005
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: POLL_ACTIVE (4), event: DEACTIVATE_RSP
    (7) disc_flags: 0x61
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: POLL_ACTIVE (4), disc_flags: 0x41
I/BrcmNfcNfa(  870): NFC received ntf gid:1
I/BrcmNfcNfa(  870): nfc_set_state 4 (IDLE)->4 (IDLE)
I/BrcmNfcNfa(  870): nfa_dm_disc_data_cback ()
I/BrcmNfcNfa(  870): nfa_dm_disc_discovery_cback (): event:0x4005
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): state: POLL_ACTIVE (4), event: DEACTIVATE_NTF
    (8) disc_flags: 0x41
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df0b5c
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): nfa_dm_disc_notify_deactivation (): activated_handle=1
I/BrcmNfcNfa(  870): nfa_dm_poll_disc_cback (): event:0x02
I/BrcmNfcNfa(  870): nfa_rw_handle_event event: NFA_RW_DEACTIVATE_NTF_EVT (0x602), flags:
    00000021
I/BrcmNfcNfa(  870): nfa_sys_ptim_stop_timer 74df45e4
I/BrcmNfcNfa(  870): ptim timer stop
I/BrcmNfcNfa(  870): Stopped presence check timer (if started)
D/BrcmNfcJni(  870): nfaConnectionCallback: event= 6
D/BrcmNfcJni(  870): nfaConnectionCallback: NFA_DEACTIVATED_EVT   Type: 0, gIsTagDeactivating:
    0
D/BrcmNfcJni(  870): NfcTag::setDeactivationState: state=0
D/BrcmNfcJni(  870): NfcTag::resetTechnologies
D/BrcmNfcJni(  870): nativeNfcTag_abortWaits
I/BrcmNfcNfa(  870): nfa_dm_disc_new_state (): old_state: POLL_ACTIVE (4), new_state: DISCOVERY
    (1) disc_flags: 0x1
I/BrcmNfcNfa(  870): nfa_dm_disc_sm_execute (): new state: DISCOVERY (1), disc_flags: 0x1
I/BrcmNfcNfa(  870): ptim timer stop
I/NfcNciHal(  870): nci_brcm_lp_timeout_cback ()
I/NfcNciHal(  870): nfc_hal_dm_power_mode_execute () event = 2
I/NfcNciHal(  870): nfc_hal_dm_set_nfc_wake () DEASSERT
D/USERIAL_LINUX(  870): UPIO_Set: ioctl, state=1
```

Apéndice D

Habilitar TRACE_LEVEL en servicio NFC

Un paso de la investigación fue ver hasta que punto se podía modificar el sistema Android para simplificar las pruebas o para depurar los servicios. Aunque el objetivo principal era conseguir implementar un ataque de *relay* con un dispositivo de serie; se desbloqueó el *bootloader* del Nexus 4 y se *rooteo* para conseguir más privilegios y poder analizar en mayor profundidad el sistema. También se estudió la posibilidad de modificar el Kernel o cargar un módulo nuevo.

D.1. Desbloqueo del terminal móvil Nexus 4

El *bootloader* es el programa que le dice al sistema operativo como encenderse; selecciona el *entry point* del Kernel y además incluye un modo *recovery* (carga de un pequeño Kernel con opciones de *reset* y actualización). Está escrito por el fabricante ya que depende del hardware sobre el que corre, y la mayoría vienen por defecto *bloqueados* para permitir únicamente el arranque de las imágenes o ROMs aprobadas por ellos, normalmente verificando que la partición de arranque no ha sido modificado mediante la comparación de una firma digital.

Aunque desbloquear el arranque no debe confundirse con el proceso de conseguir privilegios de `root`, la mayoría de veces es un requisito previo a ésto, aunque también hay casos donde explotando una vulnerabilidad[1] se puede evitar.

En cualquier caso los propios fabricantes ofrecen herramientas para desbloquear el arranque, aunque al hacerlo se pierde la garantía del dispositivo y se borran los datos almacenados, con que conviene realizar una copia de seguridad antes de iniciar el proceso.

Los pasos para llevar a cabo desbloqueo son sencillos:

1. Conectar dispositivo por USB
2. adb reboot bootloader

[1] https://towelroot.com/

3. fastboot oem unlock

Tras ésto aparecerá un mensaje de confirmación y unos segundos después de aceptar el móvil quedará desbloqueado (durante la carga del sistema se puede ver un candado abierto).

D.2. Root vía SuperSU

Para instalar la aplicación SuperSU que otorga privilegios de *root*, es necesario *flashear* el *recovery* con una imagen nueva que permite hacer las modificaciones necesarias e instalar aplicaciones, para ello existen dos opciones conocidas:

- CWM de ClockworkMod (`http://www.clockworkmod.com/rommanager`)
- TWRP de TeamWin (`http://techerrata.com/browse/twrp2/mako`)

En este caso se ha elegido la segunda, de modo que hay que descargar la última versión para `mako` (*codename* del Nexus 4).

Una vez hecho es necesario reiniciar en modo *bootloader*:

```
adb reboot bootloader
```

Se *flashea* y reinicia:

```
fastboot flash recovery openrecovery-twrp-x.x.x.x-mako.img
adb reboot
```

El siguiente paso es descargar el instalador de SuperSU[2] y traspasarlo al dispositivo, para ello se usa:

```
adb push UPDATE-SuperSU-x.zip /sdcard/
```

Se reinicia en modo *recovery* y se elige la opción "Install"buscando el fichero ZIP y confirmando. Así empezará el proceso de *flasheo*, una vez terminado el dispositivo estará *rooteado*.

También es posible iniciar una vez el *recovery* y *rootear* sin necesidad de *flashear*, claro que una vez reiniciado desaparecerá:

```
fastboot boot openrecovery-twrp-x.x.x.x-mako.img
```

Cabe mencionar que una de las características de SuperSU es que cada vez que una aplicación solicita permisos de superusuario solicita una confirmación, además de almacenar un registro con todos los eventos relativos.

[2] http://download.chainfire.eu/supersu

D.3. Flags de debug NFC

Una vez *rooteado* el dispositivo, se puede acceder a una *shell* con permisos de superusuario mediante:

```
adb shell su
```

De este modo se puede trabajar con el dispositivo del mismo modo que con la terminal de cualquier sistema Linux, aunque con algunas restricciones y bastantes menos utilidades. En cualquier caso se pueden explorar todos los directorios, ver los ficheros de configuración y listar procesos o servicios en ejecución. Durante este proceso se encontró el fichero `/etc/libnfc-brcm.conf`:

Código D.1: Fichero de configuración del sistema `libnfc-brcm.conf`.

```
########################## Start of libnfc-brcm-common.conf ##########################

###############################################################################
# Application options
APPL_TRACE_LEVEL=0x00
PROTOCOL_TRACE_LEVEL=0x00000000

###############################################################################
# performance measurement
# Change this setting to control how often USERIAL log the performance (throughput)
# data on read/write/poll
# defailt is to log performance dara for every 100 read or write
#REPORT_PERFORMANCE_MEASURE=100

###############################################################################
# File used for NFA storage
NFA_STORAGE="/data/nfc"

###############################################################################
# Snooze Mode Settings
#
#  By default snooze mode is enabled.  Set SNOOZE_MODE_CFG byte[0] to 0
#  to disable.
#
#  If SNOOZE_MODE_CFG is not provided, the default settings are used:
#  They are as follows:
#       8               Sleep Mode (0=Disabled 1=UART 8=SPI/I2C)
#       0               Idle Threshold Host
#       0               Idle Threshold HC
#       0               NFC Wake active mode (0=ActiveLow 1=ActiveHigh)
#       1               Host Wake active mode (0=ActiveLow 1=ActiveHigh)
#
#SNOOZE_MODE_CFG={08:00:00:00:01}

###############################################################################
# Insert a delay in milliseconds after NFC_WAKE and before write to NFCC
NFC_WAKE_DELAY=20

###############################################################################
# Various Delay settings (in ms) used in USERIAL
#  POWER_ON_DELAY
#     Delay after turning on chip, before writing to transport (default 300)
#  PRE_POWER_OFF_DELAY
#     Delay after deasserting NFC-Wake before turn off chip (default 0)
#  POST_POWER_OFF_DELAY
#     Delay after turning off chip, before USERIAL_close returns (default 0)
#
POWER_ON_DELAY=300
PRE_POWER_OFF_DELAY=10
#POST_POWER_OFF_DELAY=0

###############################################################################
# Maximum time (ms) to wait for RESET NTF after setting REG_PU to high
# The default is 1000.
NFCC_ENABLE_TIMEOUT=0

###############################################################################
# LPTD mode configuration
#  byte[0] is the length of the remaining bytes in this value
#     if set to 0, LPTD params will NOT be sent to NFCC (i.e. disabled).
```

```
#  byte[1] is the param id it should be set to B9.
#  byte[2] is the length of the LPTD parameters
#  byte[3] indicates if LPTD is enabled
#     if set to 0, LPTD will be disabled (parameters will still be sent).
#  byte[4-n] are the LPTD parameters.
#  By default, LPTD is enabled and default settings are used.
#  See nfc_hal_dm_cfg.c for defaults
LPTD_CFG={23:B9:21:01:02:FF:FF:04:A0:0F:40:00:80:02:02:10:00:00:00:31:0E
     :30:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00}

##############################################################################
# Startup Configuration (100 bytes maximum)
#
# For the 0xCA parameter, byte[9] (marked by 'AA') is for UICC0, and byte[10] (marked by BB) is
#    for UICC1.  The values are defined as:
#   0 : UICCx only supports ISO_DEP in low power mode.
#   2 : UICCx only supports Mifare in low power mode.
#   3 : UICCx supports both ISO_DEP and Mifare in low power mode.
#
#                                                                        AA BB
NFA_DM_START_UP_CFG={2E:CB:01:01:A5:01:01:CA:14:00:00:00:00:06:E8
     :03:00:00:00:00:00:00:00:00:00:00:00:00:00:80:01:01:C2:08:61:40:82:04:40:4B:4C:00:B5
     :03:01:02:FF}

##############################################################################
# Startup Vendor Specific Configuration (100 bytes maximum);
#  byte[0] TLV total len = 0x5
#  byte[1] NCI_MTS_CMD|NCI_GID_PROP = 0x2f
#  byte[2] NCI_MSG_FRAME_LOG = 0x9
#  byte[3] 2
#  byte[4] 0=turn off RF frame logging; 1=turn on
#  byte[5] 0=turn off SWP frame logging; 1=turn on
#  NFA_DM_START_UP_VSC_CFG={05:2F:09:02:01:01}

##############################################################################
# Antenna Configuration - This data is used when setting 0xC8 config item
# at startup (before discovery is started).  If not used, no value is sent.
#
# The settings for this value are documented here:
# http://wcgbu.broadcom.com/wpan/PM/Project%20Document%20Library/bcm20791B0/
#   Design/Doc/PHY%20register%20settings/BCM20791-B2-1027-02_PHY_Recommended_Reg_Settings.xlsx
# This document is maintained by Paul Forshaw.
#
# The values marked as ?? should be tweaked per antenna or customer/app:
# {20:C8:1E:06:??:00:??:??:??:00:??:24:00:1C:00:75:00:77:00:76:00:1C:00:03:00:0A
     :00:??:01:00:00:40:04}
# array[0] = 0x20 is length of the payload from array[1] to the end
# array[1] = 0xC8 is PREINIT_DSP_CFG
PREINIT_DSP_CFG={20:C8:1E:06:1F:00:0A:03:30:00:04:24:00:1C:00:75:00:77:00:76:00:1C:00:03:00:0A
     :00:4C:01:00:00:40:04}

##############################################################################
# Configure crystal frequency when internal LPO can't detect the frequency.
#XTAL_FREQUENCY=0

##############################################################################
# Use Nexus S NXP RC work to allow our stack/firmware to work with a retail
# Nexus S that causes IOP issues.  Note, this will not pass conformance and
# should be removed for production.
USE_NXP_P2P_RC_WORKAROUND=1

##############################################################################
# Configure the default Destination Gate used by HCI (the default is 4, which
# is the ETSI loopback gate.
#NFA_HCI_DEFAULT_DEST_GATE=0x04

##############################################################################
# Override the stack default for NFA_EE_MAX_EE_SUPPORTED set in nfc_target.h.
# The value is set to 3 by default as it assumes we will discover 0xF2,
# 0xF3, and 0xF4. If a platform will exclude and SE, this value can be reduced
# so that the stack will not wait any longer than necessary.
#NFA_MAX_EE_SUPPORTED=3

##############################################################################
# Configure the single default SE to use.  The default is to use the first
# SE that is detected by the stack.  This value might be used when the phone
# supports multiple SE (e.g. 0xF3 and 0xF4) but you want to force it to use
# one of them (e.g. 0xF4).
ACTIVE_SE=0xF4

##############################################################################
# Configure the default NfcA/IsoDep techology and protocol route. Can be
# either a secure element (e.g. 0xF4) or the host (0x00)
```

```
DEFAULT_ISODEP_ROUTE=0x00

##############################################################################
# Configure the NFC Extras to open and use a static pipe.  If the value is
# not set or set to 0, then the default is use a dynamic pipe based on a
# destination gate (see NFA_HCI_DEFAULT_DEST_GATE).  Note there is a value
# for each UICC (where F3="UICC0" and F4="UICC1")
#NFA_HCI_STATIC_PIPE_ID_F3=0x70
NFA_HCI_STATIC_PIPE_ID_F4=0x71

##############################################################################
# When disconnecting from Oberthur secure element, perform a warm-reset of
# the secure element to deselect the applet.
# The default hex value of the command is 0x3.  If this variable is undefined,
# then this feature is not used.
OBERTHUR_WARM_RESET_COMMAND=0x03

##############################################################################
# Force UICC to only listen to the following technology(s).
# The bits are defined as tNFA_TECHNOLOGY_MASK in nfa_api.h.
# Default is NFA_TECHNOLOGY_MASK_A | NFA_TECHNOLOGY_MASK_B.
UICC_LISTEN_TECH_MASK=0x01

##############################################################################
# Allow UICC to be powered off if there is no traffic.
# Timeout is in ms. If set to 0, then UICC will not be powered off.
#UICC_IDLE_TIMEOUT=30000

##############################################################################
# AID for Empty Select command
# If specified, this AID will be substituted when an Empty SELECT command is
# detected.  The first byte is the length of the AID.  Maximum length is 16.
AID_FOR_EMPTY_SELECT={08:A0:00:00:01:51:00:00:00}

##############################################################################
# Force tag polling for the following technology(s).
# The bits are defined as tNFA_TECHNOLOGY_MASK in nfa_api.h.
# Default is NFA_TECHNOLOGY_MASK_A | NFA_TECHNOLOGY_MASK_B |
#             NFA_TECHNOLOGY_MASK_F | NFA_TECHNOLOGY_MASK_ISO15693 |
#             NFA_TECHNOLOGY_MASK_B_PRIME | NFA_TECHNOLOGY_MASK_A_ACTIVE |
#             NFA_TECHNOLOGY_MASK_F_ACTIVE.
#
# Notable bits:
#   NFA_TECHNOLOGY_MASK_KOVIO         0x20
#   NFA_TECHNOLOGY_MASK_A_ACTIVE      0x40
#   NFA_TECHNOLOGY_MASK_F_ACTIVE      0x80
POLLING_TECH_MASK=0xFF

##############################################################################
# Force P2P to only listen for the following technology(s).
# The bits are defined as tNFA_TECHNOLOGY_MASK in nfa_api.h.
# Default is NFA_TECHNOLOGY_MASK_A | NFA_TECHNOLOGY_MASK_F |
#             NFA_TECHNOLOGY_MASK_A_ACTIVE | NFA_TECHNOLOGY_MASK_F_ACTIVE
P2P_LISTEN_TECH_MASK=0xC5

##############################################################################
# Maximum Number of Credits to be allowed by the NFCC
#   This value overrides what the NFCC specifices allowing the host to have
#   the control to work-around transport limitations.  If this value does
#   not exist or is set to 0, the NFCC will provide the number of credits.
#MAX_RF_DATA_CREDITS=1

##############################################################################
# This setting allows you to disable registering the T4t Virtual SE that causes
# the NFCC to send PPSE requests to the DH.
# The default setting is enabled (i.e. T4t Virtual SE is registered).
#REGISTER_VIRTUAL_SE=1

##############################################################################
# When screen is turned off, specify the desired power state of the controller.
# 0: power-off-sleep state; DEFAULT
# 1: full-power state
# 2: screen-off card-emulation (CE4/CE3/CE1 modes are used)
#SCREEN_OFF_POWER_STATE=0

##############################################################################
# Firmware patch file
#  If the value is not set then patch download is disabled.
FW_PATCH="/vendor/firmware/bcm2079x_firmware.ncd"

##############################################################################
# Firmware pre-patch file (sent before the above patch file)
#  If the value is not set then pre-patch is not used.
```

```
FW_PRE_PATCH="/vendor/firmware/bcm2079x_pre_firmware.ncd"

##############################################################################
# Firmware patch format
#    1 = HCD
#    2 = NCD (default)
#NFA_CONFIG_FORMAT=2

##############################################################################
# SPD Debug mode
#  If set to 1, any failure of downloading a patch will trigger a hard-stop
#SPD_DEBUG=0

##############################################################################
# SPD Max Retry Count
#  The number of attempts to download a patch before giving up (defualt is 3).
#  Note, this resets after a power-cycle.
#SPD_MAX_RETRY_COUNT=3

##############################################################################
# transport driver
#
# TRANSPORT_DRIVER=<driver>
#
#  where <driver> can be, for example:
#     "/dev/ttyS"          (UART)
#     "/dev/bcmi2cnfc"     (I2C)
#     "hwtun"              (HW Tunnel)
#     "/dev/bcmspinfc"     (SPI)
#     "/dev/btusb0"        (BT USB)
TRANSPORT_DRIVER="/dev/bcm2079x-i2c"

##############################################################################
# power control driver
# Specify a kernel driver that support ioctl commands to control NFC_EN and
# NFC_WAKE gpio signals.
#
# POWER_CONTRL_DRIVER=<driver>
#  where <driver> can be, for example:
#     "/dev/nfcpower"
#     "/dev/bcmi2cnfc"     (I2C)
#     "/dev/bcmspinfc"     (SPI)
#     i2c and spi driver may be used to control NFC_EN and NFC_WAKE signal
POWER_CONTROL_DRIVER="/dev/bcm2079x-i2c"

##############################################################################
# I2C transport driver options
# Mako does not support 10-bit I2C addresses
# Revert to 7-bit address
BCMI2CNFC_ADDRESS=0x77

##############################################################################
# I2C transport driver try to read multiple packets in read() if data is available
# remove the comment below to enable this feature
#READ_MULTIPLE_PACKETS=1

##############################################################################
# SPI transport driver options
#SPI_NEGOTIATION={0A:F0:00:01:00:00:00:FF:FF:00:00}

##############################################################################
# UART transport driver options
#
# PORT=1,2,3,...
# BAUD=115200, 19200, 9600, 4800,
# DATABITS=8, 7, 6, 5
# PARITY="even" | "odd" | "none"
# STOPBITS="0" | "1" | "1.5" | "2"

#UART_PORT=2
#UART_BAUD=115200
#UART_DATABITS=8
#UART_PARITY="none"
#UART_STOPBITS="1"

##############################################################################
# Insert a delay in microseconds per byte after a write to NFCC.
# after writing a block of data to the NFCC, delay this an amopunt of time before
# writing next block of data.  the delay is calculated as below
#    NFC_WRITE_DELAY * (number of byte written) / 1000 milliseconds
# e.g. after 259 bytes is written, delay (259 * 20 / 1000) 5 ms before next write
NFC_WRITE_DELAY=20
```

```
##############################################################################
# Default poll duration (in ms)
#  The defualt is 500ms if not set (see nfc_target.h)
#NFA_DM_DISC_DURATION_POLL=333
```

Este fichero, es leído cada vez que se inicia el servicio NFC y permite realizar algunos cambios interesantes sin necesidad de recompilar la librería, lo que ahorra bastante trabajo. Uno de los cambios que se llevo a cabo fue el del parámetro `APPL_TRACE_LEVEL`, poniendo el valor a `0x05` se consigue que toda la librería muestre mensajes de depuración facilitando la traza de la ejecución.

Si bien es cierto, que para modificar el fichero surgen algunos problemas:

1. El fichero se encuentra en `/etc` que es un enlace simbólico a `/system/etc` y por defecto es una partición montada como sólo lectura (si se mira el fichero `fstab.mako` o `/proc/mounts` se encuentra la opción `ro`, por lo que es necesario desmontarla y volverla a montar (desde una *shell root*):

   ```
   mount -o remount rw /system
   ```

   Una vez hecho ya se puede escribir en la partición.

2. La *shell* de Android no tiene muchos programas como `vi`, `ed` o `grep`, con que editar un fichero se puede convertir en una tarea difícil, la solución es pasar el fichero a nuestro PC, editarlo cómodamente y volver a cargarlo:

   ```
   adb pull /etc/libnfc-brcm.conf .
   adb push libnfc-brcm.conf.edit /etc/libnfc-brcm.conf
   ```

3. Pero da un error porque la orden `push` no tiene permisos *root* para escribir en `/etc`, la solución es copiarlo temporalmente a `/sdcard` donde si se tiene permiso y posteriormente moverlo:

   ```
   adb push libnfc-brcm.conf.edit /sdcard/.
   ```

   Y desde una consola con permisos de superusuario (`mv` fallaría al ser una referencia *cross-device*):

   ```
   cp /sdcard/libnfc-brcm.conf.edit /etc/libnfc-brcm.conf && rm /
       sdcard/libnfc-brcm.conf
   ```

   Otra opción, por poder hacer las cosas "de otra formaçonsiste en usar `cat` y redireccionar la salida:

   ```
   cat /sdcard/libnfc-brcm.conf.edit > /etc/libnfc-brcm.conf
   ```

   Ahora tan sólo hay que reiniciar el servicio NFC y conectar el dispositivo por USB para ejecutar la orden `adb logcat` y empezar a ver los registros.

En las pruebas se obtuvieron trazas del inicio y la parada del servicio (donde se comprobó, apoyándose en el código fuente, que el fichero de configuración era cargado, junto con el *firmware* y las comprobaciones de actualizaciones, además de ver las notificaciones

y mensajes de configuración desde el DH al NFCC), también se obtuvieron trazas de las transacciones de lectura y escritura de un *tag* remoto así como del proceso de *discovery*.

Las trazas resultaron de gran ayuda para la comprensión de la implementación de la NCI.

Apéndice E

Kernel Android

Durante el desarrollo de este proyecto se han comprobado los requisitos y factibilidad para compilar y distribuir un módulo para el Kernel, en caso de que fuera necesario intervenir el sistema a ese nivel. Aquí se documentan los distintos Kernels de Android y como compilar un módulo para uno específico, para finalmente determinar algunas ventajas y desventajas.

Android está basado en el Kernel de Linux, aunque a estas alturas poco tiene que ver uno con el otro. Entre otras cosas ha sido modificado para implementar IPC (*Inter Process Communication*) y ahorrar batería, permite también ejecutar las aplicaciones en un entorno aislado, etc.

Sin embargo, aunque están basados en el mismo, los distintos fabricantes utilizan distintas ramas con soporte para su hardware específico, por lo que a la hora de compilar un módulo nuevo es necesario trabajar con la rama correspondiente al Kernel utilizado en el dispositivo destino. Como casi todo en Android, el código fuente es libre y se puede encontrar en el git `https://android.googlesource.com/device/lge/mako-kernel/` (las URLs son de la forma "`device/<vendor>/<name>`"), además en la documentación se encuentran la Tabla E.1 se pueden ver las principales ramas.

En el caso de Nexus 4, dispositivo con el que se han hecho las pruebas, el codename es `mako` con que es necesario clonar el repositorio:

```
$ git clone https://android.googlesource.com/kernel/msm.git destino
```

Una vez descargado se va a compilar el Kernel, para ello es necesario el Android Toolchain para compilar para la plataforma deseada (arm, x86 o mips), ya que otra de las características de Android es que puede correr en diferentes arquitecturas, de modo que se debe descargarse la versión adecuada para la máquina host y añadir la ruta al PATH. El siguiente paso será compilar:

```
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=arm-linux-androideabi-
make mako_defconfig
make
```

Device	Binary location	Source	Build config
hammerhead	device/lge/hammerhead-kernel	kernel/msm	hammerhead_defconfig
flo	device/asus/flo-kernel/kernel	kernel/msm	flo_defconfig
deb	device/asus/flo-kernel/kernel	kernel/msm	flo_defconfig
manta	device/samsung/manta/kernel	kernel/exynos	manta_defconfig
mako	device/lge/mako-kernel/kernel	kernel/msm	mako_defconfig
grouper	device/asus/grouper/kernel	kernel/tegra	tegra3_android_defconfig
tilapia	device/asus/grouper/kernel	kernel/tegra	tegra3_android_defconfig
maguro	device/samsung/tuna/kernel	kernel/omap	tuna_defconfig
toro	device/samsung/tuna/kernel	kernel/omap	tuna_defconfig
panda	device/ti/panda/kernel	kernel/omap	panda_defconfig
stingray	device/moto/wingray/kernel	kernel/tegra	stingray_defconfig
wingray	device/moto/wingray/kernel	kernel/tegra	stingray_defconfig
crespo	device/samsung/crespo/kernel	kernel/samsung	herring_defconfig
crespo4g	device/samsung/crespo/kernel	kernel/samsung	herring_defconfig

Tabla E.1: Nombres y localización de los principales kernels Android.

Es necesario definir las variables de la arquitectura destino y el prefijo del *toolchain*, y en el ejemplo se utiliza la configuración por defecto, pero es posible muchos parámetros.

Una vez se ha compilado el Kernel, la forma más sencilla es compilar el módulo solo:

1. Se crea un directorio con los ficheros fuentes del Kernel.
2. Se crea un Makefile para compilar el módulo linkeando con el Kernel:

```
KERNEL_DIR=<ruta_del_kernel>

obj-m := <objecto_salida>
PWD := $(shell pwd)
default:
	$(MAKE) ARCH=arm CROSS_COMPILE=arm-linux-androideabi- -C
	    $(KERNEL_DIR) SUBDIRS=$(PWD) modules
clean:
	$(MAKE) -C $(KERNEL_DIR) SUBDIRS=$(PWD) clean
```

3. Se ejecuta el `make`. Y si todo ha ido bien aparecerá el módulo `modulo.ko` que podrá cargarse con la orden `insmod`.

Aunque también sería posible añadir el módulo al directorio de módulos y modificar el Makefile para llevar a cabo una compilación estática.

Por desgracia, la mayoría de versiones, incluida ésta, por defecto no incluyen soporte para módulos (para ello es necesario recompilar con el *flag* `CONFIG_MODULES=y`). Lo que significa que para que alguien utilice un módulo necesita recompilar el Kernel, ya sea con el *flag* y después cargando el módulo o con el módulo estático, en cualquier caso es un proceso poco usable, y al igual que utilizar una ROM modificada rompe el requisito de utilizar software lo más genérico posible.

En cualquier caso para probar el módulo, habría que seguir los siguientes pasos:

1. Compilar un Kernel con soporte para módulos, *flashear* el teléfono con la nueva imagen y cargar el módulo:

```
make bootimage
fastboot flash boot zImage.img
fastboot reboot
adb push module.ko /sdcard/.
adb shell su insmod /sdcard/module.ko
```

2. Compilar el Kernel y en lugar de *flashear*, usar `fastboot` para cargar la nueva imagen sólo una vez, subir el módulo y cargarlo:

```
make bootimage
fastboot boot zImage.img
fastboot reboot
adb push module.ko /sdcard/.
adb shell su insmod /sdcard/module.ko
```

3. Lanzar el Kernel con el emulador de Android y cargar el módulo.

```
make bootimage
emulator -avd foo -kernel zImage.img
adb push module.ko /sdcard/.
adb shell su insmod /sdcard/module.ko
```

La segunda opción seguramente sea la más práctica y permite probar correctamente cuando se trata con elementos difíciles de emular, pero en muchas ocasiones puede ser peligroso hacer pruebas sobre el Kernel en un dispositivo por lo que emular es la solución más adecuada. En Android, la rama de Kernels preparada para soportar emulación es Goldfish, por lo que será necesario compilar con ese Kernel los módulos que se quieran emular.

Apéndice F

Estructura y órdenes APDU

Orden APDU		
Campo	**Long. (bytes)**	**Descripción**
CLA	1	Clase - indica el tipo de orden, p.e: estándar o propietario
INS	1	Código - indica la orden específica, p.e: "escribir"
P1-P2	2	Parámetros de la orden, p.e: offset en el que escribir
L_c	0,1 or 3	Codifica la longitud (N_c) en bytes de los datos
Datos	N_c	N_c bytes de datos
L_e	0,1,2 or 3	Codifica la lognitud máxima (N_e) en bytes para la respuesta
Respuesta APDU		
Datos	$N_r \leq N_e$	Datos de respuesta
SW1-SW2 (trailer)	2	Estado de la orden procesada, p.e: 0x90 0x00 indica éxito

Tabla F.1: Estructura de mensajes APDU: orden-respuesta.

Tabla F.2: Órdenes para smart-cards más comúnes. Extraído de `http://techmeonline.com/most-used-smart-card-commands-apdu/`.

Command	Function	INS	Standard
ACTIVATE FILE	Reversibly unblock a file.	'44'	ISO/IEC 7816-9
APPEND RECORD	Insert a new record in a file with a linear fixed structure.	'E2'	ISO/IEC 7816-4
APPLICATION BLOCK	Reversibly block an application.	'1E'	EMV
APPLICATION UNBLOCK	Unblock an application.	'18'	EMV
ASK RANDOM	Request a random number from the smart card.	'84'	EN 726-3
CHANGE CHV	Change the PIN.	'24'	TS 51.011
CHANGE REFERENCE DATA	Change the data used for user identification (e.g., a PIN).	'24'	ISO/IEC 7816-8
CLOSE APPLICATION	Reset all attained access condition levels.	'AC'	EN 726-3
CONVERT IEP CURRENCY	Convert currency.	'56'	EN 1546-3
CREATE FILE	Create a new file.	'E0'	ISO/IEC 7816-9
CREATE RECORD	Create a new record in a record-oriented file.	'E2'	EN 726-3
CREDIT IEP	Load the purse (IEP).	'52'	EN 1546-3
CREDIT PSAM	Pay from IEP to the PSAM.	'72'	EN 1546-3
DEACTIVATE FILE	Reversibly block a file.	'04'	ISO/IEC 7816-9
DEBIT IEP	Pay from the purse	'54'	EN 1546-
DECREASE	Reduce the value of a counter in a file.	'30'	EN 726-3
DECREASE STAMPED	Reduce the value of a counter in a file that is protected using a cryptographic checksum.	'34'	EN 726-3
DELETE	Delete a uniquely identifiable object (such as a load file, application or key).	'E4'	OP
DELETE FILE	Delete a file.	'E4'	ISO/IEC 7816-9
DISABLE CHV	Disable PIN queries.	'26'	TS 51.011
DISABLE VERIFICATION REQUIREMENT	Disable user identification (e.g., PIN queries).	'26'	ISO/IEC 7816-8
ENABLE CHV	Enable PIN queries.	'28'	TS 51.011, EN 726-3

ENABLE VERIFICATION REQUIREMENT	Enable user identification (e.g., PIN queries).	'28'	ISO/IEC 7816-8
ENVELOPE	Embed a command in a smart card command.	'C2'	EN 726-3, ISO/IEC 7816-4
ERASE BINARY	Set the content of a file with a transparent structure to the erased state.	'0E'	ISO/IEC 7816-4
EXECUTE	Execute a file.	'AE'	EN 726-3
EXTEND	Extend a file.	'D4'	EN 726-3
EXTERNAL AUTHENTICATE	Authenticate the outside world with respect to the smart card.	'82'	ISO/IEC 7816-4
GENERATE AUTHORISATION CRYPTOGRAM	Generate a signature for a payment transaction.	'AE'	EMV
GENERATE PUBLIC KEY PAIR	Generate a key pair for an asymmetric cryptographic algorithm.	'46'	ISO/IEC 7816-8
GET CHALLENGE	Request a random number from the smart card.	'84'	ISO/IEC 7816-4
GET DATA	Read TLV-coded data objects.	'CA'	ISO/IEC 7816-4
GET PREVIOUS IEP SIGNATURE	Repeat the computation and output of the last signature received IEP.	'5A'	EN 1546-3
GET PREVIOUS PSAM SIGNATURE	Repeat the computation and output of the last signature received from the PSAM.	'86'	EN 1546-3
GET RESPONSE	Request data from the smart card (used with the T = 0 transmission protocol).	'C0'	TS 51.011
GET STATUS	Read the life-cycle state information of the card manager, application and load file.	'F2'	ISO/IEC 7816-4, OP
GIVE RANDOM	Send a random number to the smart card.	'86'	EN 726-3
INCREASE	Increase the value of a counter in a file.	'32'	TS 51.011
INCREASE STAMPED	Increase the value of a counter in a file that is protected using a cryptographic checksum.	'36'	EN 726-3

INITIALIZE IEP	Initialize IEP for a subsequent purse command.	'50'	EN 1546-3
INITIALIZE PSAM	Initialize PSAM for a subsequent purse command.	'70'	EN 1546-3
INITIALIZE PSAM for Offline Collection	Initialize PSAM for offline booking of the amount.	'7C'	EN 1546-3
INITIALIZE PSAM for Online Collection	Initialize PSAM for online booking of the amount.	'76'	EN 1546-3
INITIALIZE PSAM for Update INSTALL INTERNAL AUTHENTICATE	Initialize PSAM for changing the parameters. Install an application by invoking various oncard functions of the card manager and/or security domain. Authenticate the smart card with respect to the outside world.	'80' 'E6' '88'	EN 1546-3, OP, ISO/IEC 7816-4
INVALIDATE	Reversibly block a file.	'04'	TS 51.011, EN 726-3
ISSUER AUTHENTICATE	Verify a signature of the card issuer.	'82'	EMV-2
LOAD	Load an application by transferring the load file.	'E8'	OP
??? LOAD KEY FILE LOCK MANAGE CHANNEL	Load keys in files using cryptographic protection. Irreversibly block a file. Control the logical channels of a smart card.	'D8' '76' '70'	EN 726-3, EN 726-3 ISO/IEC 7816-4
??? LOAD KEY FILE LOCK MANAGE CHANNEL	Load keys in files using cryptographic protection. Irreversibly block a file. Control the logical channels of a smart card.	'D8' '76' '70'	EN 726-3, EN 726-3 ISO/IEC 7816-4
MANAGE SECURITY ENVIRONMENT	Change the parameters for using cryptographic algorithms in the smart card.	'22'	ISO/IEC 7816-8
MUTUAL AUTHENTICATE	Mutually authenticate the smart card and the terminal.	'82'	ISO/IEC 7816-8
PERFORM SCQL OPERATION	Execute an SCQL instruction.	'10'	ISO/IEC 7816-7

PERFORM SECURITY OPERATION	Execute a cryptographic algorithm in the smart card.	'2A'	ISO/IEC 7816-8
PERFORM TRANSACTION OPERATION	Execute an SCQL transaction instruction.	'12'	ISO/IEC 7816-7
PERFORM USER OPERATION	Manage users in the context of SCQL.	'14'	ISO/IEC 7816-7
PSAM COLLECT	Execute PSAM online booking of an amount.	'78'	EN 1546-3
PSAM COLLECT	End PSAM online booking of an amount.	'7A'	EN 1546-3
PSAM COMPLETE	End paying the IEP against the PSAM.	'74'	EN 1546-3
PSAM VERIFY COLLECTION	End PSAM offline booking of an amount.	'7E'	EN 1546-3
PUT DATA	Write TLV-coded data objects.	'DA'	ISO/IEC 7816-4
PUT KEY	Write one or more new keys or replace existing keys.	'D8'	OP
REACTIVATE FILE	Unblock a file.	'44'	ISO/IEC 7816-9
READ BINARY	Read from a file with a transparent structure.	'B0'	TS 51.011
READ BINARY STAMPED	Read data from a file with a transparent structure that is secured with a cryptographic checksum.	'B4'	ISO/IEC 7816-4
READ RECORD	Read data from a file with a record-oriented structure.	'B2'	TS 51.011
READ RECORD(S)	Read data from a file with a record-oriented structure.	'B2'	ISO/IEC 7816-4
READ RECORD STAMPED	Read data from a file with a record-oriented structure that is secured with a cryptographic checksum.	'B6'	EN 726-3
REHABILITATE	Unblock a file.	'44'	TS 51.011 EN ???
RESET RETRY COUNTER	Reset an error counter.	'2C'	ISO/IEC 7816-8
RUN GSM ALGORITHM	Execute a GSM-specific cryptographic algorithm.	'88'	TS 51.011
SEARCH BINARY	Search for a text string in a file with a transparent structure.	'A0'	ISO/IEC 7816-9

SEARCH RECORD	Search for a text string in a file with a record-oriented structure.	'A2'	ISO/IEC 7816-9
SEEK	Search for a text string in a file with a record-oriented structure.	'A2'	TS 51.011, EN 726-3
SELECT	Select a file.	'A4'	TS 51.011
SELECT (FILE)	Select a file.	'A4'	ISO/IEC 7816-4
SET STATUS	Write life-cycle state data for the card manager, application and load file.	'F0'	OP
SLEEP	Obsolete command for setting the smart card in a power-saving state.	'FA'	TS 51.011
STATUS	Read various data from the currently selected file.	'F2'	TS 51.011
TERMINATE CARD USAGE	Irreversibly block a smart card.	'FE'	ISO/IEC 7816-9
TERMINATE DF	Irreversibly block a DF.	'E6'	ISO/IEC 7816-9
TERMINATE EF	Irreversibly block an EF.	'E8'	ISO/IEC 7816-9
UNBLOCK CHV	Reset a PIN retry counter that has reached its maximum value.	'2C'	TS 51.011 EN
UPDATE BINARY	Write to a file with a transparent structure.	'D6'	TS 51.011, ISO/IEC7816-4
UPDATE IEP PARAMETER	Change the general parameters of a purse.	'58'	EN 1546-3
UPDATE PSAM Parameter (offline)	Modify the parameters in the PSAM (offline).	'84'	EN 1546-3
UPDATE PSAM Parameter (online)	Modify the parameters in the PSAM (online).	'82'	EN 1546-3
UPDATE RECORD	Write to a file with a linear fixed, linear variable or cyclic structure.	'DC'	TS 51.011, ISO/IEC 7816-4
VERIFY	Verify the transferred data (such as a PIN).	'20'	ISO/IEC 7816-4, EMV
VERIFY CHV	Verify the PIN.	'20'	TS 51.011
WRITE BINARY	Write to a file with a transparent structure using a logical AND/OR process.	'D0'	ISO/IEC 7816-4
WRITE RECORD	Write to a file with a record-oriented structure using a logical AND/OR process.	'D2'	ISO/IEC 7816-4

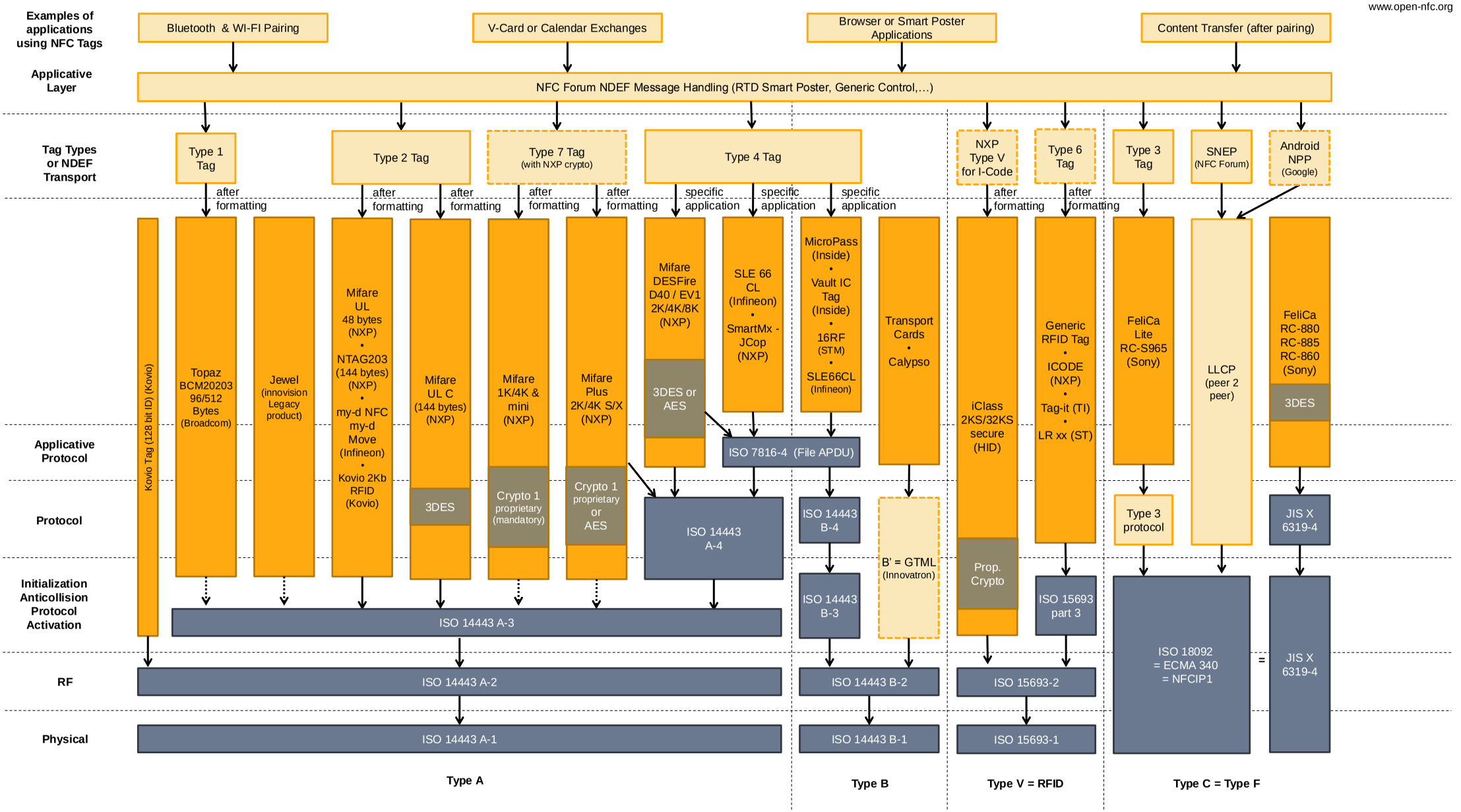
Apéndice G

Mapa de *tags* y tecnologías

NFC Standards, Products and Specifications

Version 1.9
jbblanchet@insidefr.com

www.open-nfc.org



Standards | Specifications | Proprietary Specifications

Cryptography | Examples of Product

rely on

proprietary mapping, not 100% compliant with the specification